VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Detekce průstřelů v terčích

# Detection of Bullet Holes in Targets

2020 Marek Kneys

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Marek Kneys**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Detekce průstřelů v terčích
Detection of Bullet Holes in Targets

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je detekovat průstřely v terčích typu pistolový mezinárodní terč 50/20 pomocí metod analýzy obrazu. Aplikace bude fungovat na mobilních telefonech a výstupem bude počet nastřílených bodů.

Ve své práci proveďte:
1. Seznamte se s možnými postupy detekce průstřelů v terčích.
2. Naimplementujte aplikaci pro prostředí Android s využitím knihovny OpenCV.
3. Experimentálně ověřte přesnost a rychlost navrženého řešení.
4. Své závěry zdokumentuje v textu práce.

Seznam doporučené odborné literatury:

[1] P. R. Aryan, "Vision based automatic target scoring system for mobile shooting range," 2012 International Conference on Advanced Computer Science and Information Systems (ICACSIS), Depok, 2012, pp. 325-329.
[2] Y. Lin, S. Miaou, Y. Lin and S. Chen, "An automatic scoring system for air pistol shooting competition based on image recognition of target sheets," 2015 IEEE International Conference on Consumer Electronics - Taiwan, Taipei, 2015, pp. 140-141.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Michael Holuša, Ph.D.**

Datum zadání: 01.09.2019
Datum odevzdání: 30.04.2020

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky 1D16 OSTRAVA

doc. Ing. Jan Platoš, Ph.D.
*vedoucí katedry*

prof. Ing. Pavel Brandštetter, CSc.
*děkan fakulty*

# Prohlášení studenta

Prohlašuji, že jsem diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: 15. května 2020

.....
podpis studenta

Rád bych poděkoval vedoucímu diplomové práce Ing. Michael Holuša, Ph.D. za odbornou pomoc a konzultaci při vytváření této práce.

**Abstrakt**

Tato diplomová práce pojednává o návrhu a implementaci mobilní aplikace s operačním systémem Android, jejíž úkol je detekovat průstřely v pistolových mezinárodních terčích 50/20 pomocí kamery na telefonu. Řešení je postaveno na metodách analýzy obrazu a konvolučních neuronových sítích. Na základě detekovaných průstřelů, transformace terče do jednotné podoby a vhodné šablony s informací o rozložení hodnot bodů v terči, bude aplikace automaticky bodovat střelce.

**Klíčová slova**: Android aplikace, konvoluční neuronové sítě, průstřel, pistolový mezinárodní terč 50/20

**Abstract**

This diploma thesis deals with design and implementation of mobile application with Android operating system, whose task is to detect bullets in International pistol targets 50/20 using camera on phone. The solution is based on image analysis methods and convolutional neural networks. Android application will automatically score the shooter based on detected bullet holes, transformation of the target into normalized shape and using a template with information about the distribution of point values in the target.

**Key Words**: Android application, convolution neural networks, bullet hole, international pistol target 50/20

# Obsah

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| CNN | – | Convolution neural networks |
| YOLO CNN | – | You look only once convolution neural network |
| HSV | – | Hue saturation value color model |
| FPS | – | Frames per second |
| Boobs | – | YOLO bounding box Annotation Tool |
| R-CNN | – | Region - Convolution neural network |
| SVM | – | Support vector machine |
| MAP | – | Mean average precision |
| IOU | – | Intersection over union |
| HOG | – | Histogram of oriented gradients |
| MMOD | – | Max margin object detection |
| IOU | – | Intersection over union |
| TP | – | True positives |
| FP | – | False positives |
| FN | – | False negatives |
| NMS | – | Non-maximum Suppression |
| IOS | – | iPhone operation system |
| JVM | – | Java virtual machine |
| NDK | – | Native development kit |
| JNI | – | Java native interface |

# Seznam obrázků

# Seznam tabulek

# 1 Úvod

V tomto dokumentu bude podrobně popsán postup implementace Android mobilní aplikace, pomocí níž by měl být střelec automaticky bodován.

Základní motivací pro vznik takovéto aplikace je ulehčit střelci počítání bodů z průstřelů v terči. Aplikace by mohla sloužit civilním střelcům nebo samotným střelnicím, organizující nějaké střelecké soutěže. Další použití vidím například v armádě nebo u policie, při trénování střelby na cíl.

Aplikace byla testována hlavně na pistolovém mezinárodním terči 50/20 s průstřely vytvořenými projektily ráží 9 mm, nicméně byla navržena tak, aby mohla být později jednoduše rozšířena i pro práci s jinými typy terčů.

Text se celkově skládá z pěti hlavních částí, kterými jsou: vysegmentování terče z obrazu a jeho transformace do jednotné velikosti a tvaru, část zabývající se samotnou detekcí průstřelů pomocí konvoluční neuronové sítě, způsob bodování detekovaných průstřelů na základě normalizované terčové šablony, použité technologie pro implementaci, popis architektury a implementace aplikace v operačním systému Android.

První částí celého procesu je vysegmentování terče z obrazu a transformování terče do jednotné velikosti a tvaru, tak aby co nejvíce odpovídal terčové šabloně, která obsahuje informace o distribuci bodových hodnot v terči. Tento postup také omezí potenciální pozitivně negativní detekce konvoluční neuronové sítě mimo terč, jelikož se tímto krokem kompletně zbavíme pozadí terče. Na konci této částí je zhodnocení celého procesu.

Část zabývající se detektorem průstřelů a bodováním pojednává o vytvoření datové sady průstřelů pro trénování sítě, srovnání některých možných metod detekce objektů, podrobný popis vybraného typu detektoru založený na implementaci tzv. YOLO [11] konvoluční neuronové sítě a samotném procesu učení této sítě. Na konci této částí je zhodnocení detektoru a jeho úspěšnosti.

Další část pojednává o způsobu bodování detekovaných průstřelů na základě normalizované terčové šablony, také zde rozebírám možné jiné alternativní způsoby bodování a popíšu případy, kdy by mohl použitý způsob selhat.

V další části popisuji vybrané technologie pro implementaci aplikace a zdůvodňuji proč jsem si vybral zrovna je.

Poslední důležitou částí je rozbor architektury a přesný popis, jak mezi sebou jednotlivé oddělené komponenty komunikují. Tato kapitola obsahuje také vizualizaci jednoduchého uživatelského rozhraní.

Na konci dokumentu celkově zhodnotím aplikaci, popíšu její klady a zápory a navrhnu postup, který by mohl aplikaci v budoucnu zlepšit.

# 2 Segmentace terče z obrazu a jeho transformace do jednotné velikosti a tvaru

Tato kapitola se dělí na dvě části, kterými jsou, popis implementace segmentace terče z obrázku a jeho transformace do jednotné podoby a zhodnocení tohoto procesu.

## 2.1 Popis implementace

Zde podrobně rozebírám, jakým způsobem extrahuji terč z obrazu a jak ho transformuji do normalizované podoby. Řešení spočívá v hranové analýze obrazu, pomocí níž segmentuji obraz na jednotlivé objekty oddělené hranami. Z nich podle níže popsaných kritérií vyberu objekt terče, naleznu jeho rohy a perspektivní transformací ho transformuji do nového obrazu. Tento obraz následně znovu otestuji, jestli je to opravdu terč. Pokud terč projde testem, segmentace proběhla úspěšně, pokud neprojde, aplikace si vyžádá další snímek z kamery. Níže budou podrobně popsány dílčí kroky extrakce terče.

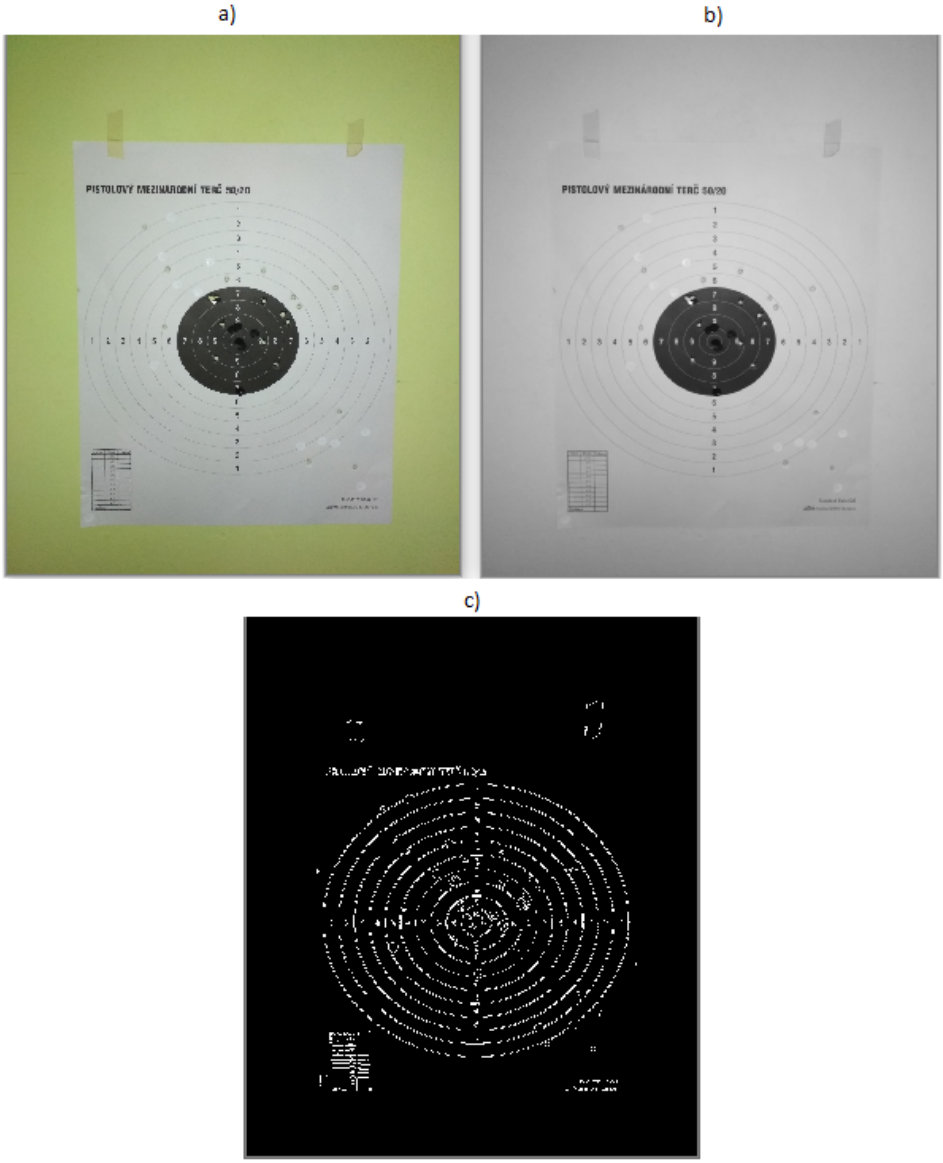
### 2.1.1 Úprava velikosti vstupního obrazu a následná hranová analýza

Po pořízení snímku terče a jeho okolí jsem kvůli omezenému výpočetnímu výkonu na mobilním zařízení musel omezit maximální velikost dimenze obrazu. Nastavil jsem tedy tuto konstantu na hodnotu 1300 pixelů (Konstanta byla zjištěna experimentálně při testování výkonu aplikace). Pokud je snímek v alespoň jedné dimenzi větší než tato maximální hodnota, zmenším ho, tak aby se maximální hodnota dimenze co nejvíce ze spodu blížila maximální hodnotě při zachování poměru výšky a šířky. Při zmenšování obrazu jsem použil bilineární interpolaci.

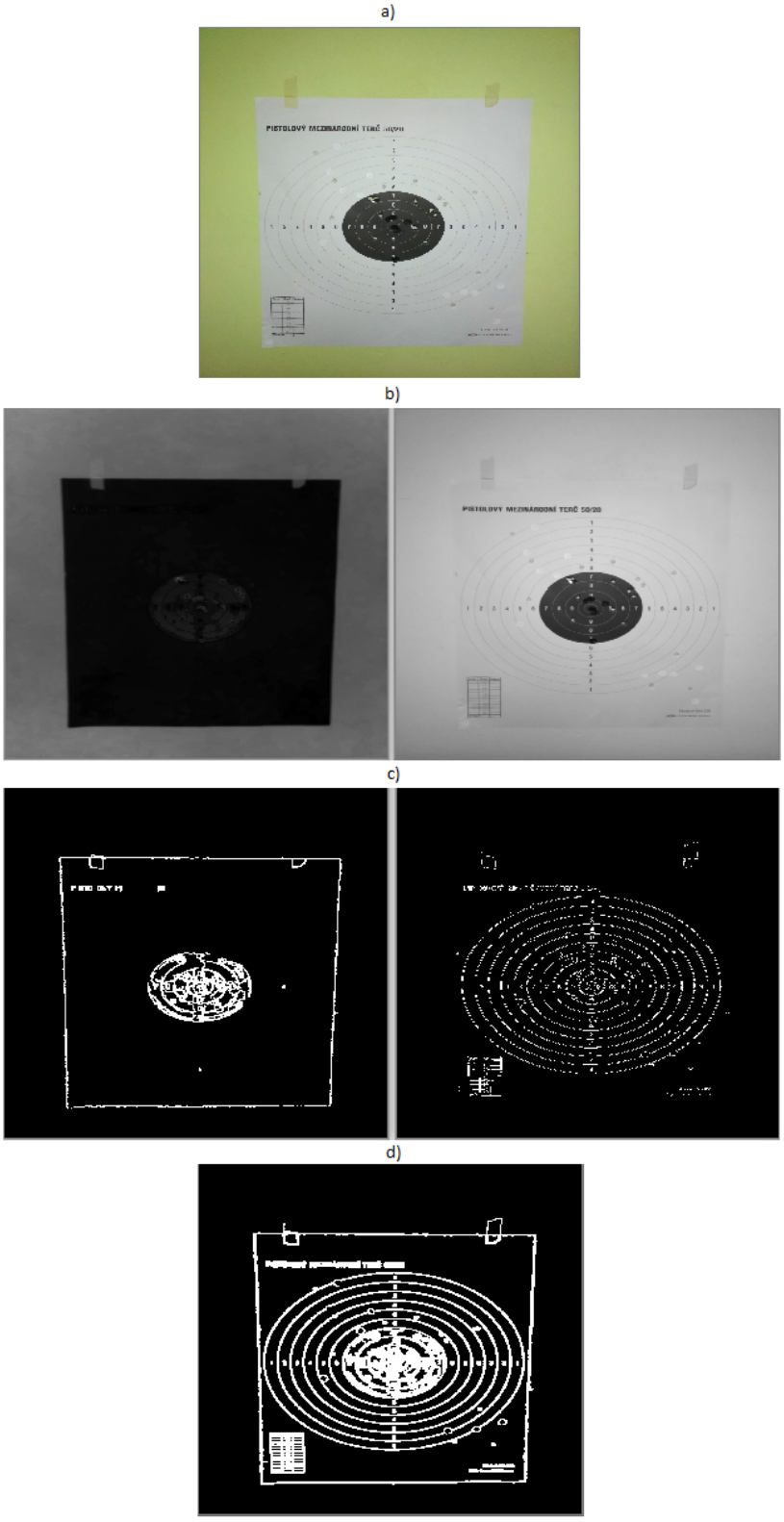
Dalším krokem je dostat ze vstupního obrazu informace o hranách, pomocí kterých vysegmentuji terč z obrazu. Prvním krokem je potlačení šumu v obrazu, tak aby hranová analýza byla co nejpřesnější, pro tento účel používám aplikaci takzvaného box blur filtru [1], který v každém pixelu obrazu spočítá průměr hodnot okolních pixelů a na pozici počítaného pixelu ji uloží do nového obrazu.

Po odstranění šumu jsem mohl přistoupit k samotné extrakci hran, kterou jsem se nejprve snažil dělat na obrazu, který jsem převedl do odstínů šedi a zbavil jsem se tak informace o barvě. Zde jsem ale později narazil na problém, když jsem měl terč a pozadí s jinou barvou a jeho hrany tím pádem nesplývaly s okolím, nicméně pokud mělo pozadí blízko hran terče podobnou nebo stejnou intenzitu, hrana se zde neobjevila, což následně celou extrakci terče znemožnilo. Tuto situaci můžeme vidět na obrázku 1. Musel jsem tedy začít počítat i s barvou pozadí, jelikož obrázek ve stupních šedi nestačil. Pro tento účel jsem obraz převedl do HSV [2] reprezentace obrazu. Tato reprezentace se skládá ze tří kanálů, Hue (odstín), Saturační a Jasový kanál. Použil jsem informace ze Saturačního a Jasového kanálu, které jsem pak zkombinoval. Saturační kanál nese informace právě o sytosti barvy a Jasový kanál určuje světlost barvy v obrazu.

Pro tyhle dva kanály udělám hranovou analýzu pomocí Cannyho detekce hran [3] a výsledky zkombinuji, tak aby celkový výsledek hranové analýzy obsahoval hrany jak z prvního kanálu tak z druhého. Nakonec aplikuji morfologickou operaci dilatace [4] s velikostí filtru 4 x 4, abych vyplnil malé potenciální díry v hranici terče. Celý postup je vizualizován na obrázku 2.



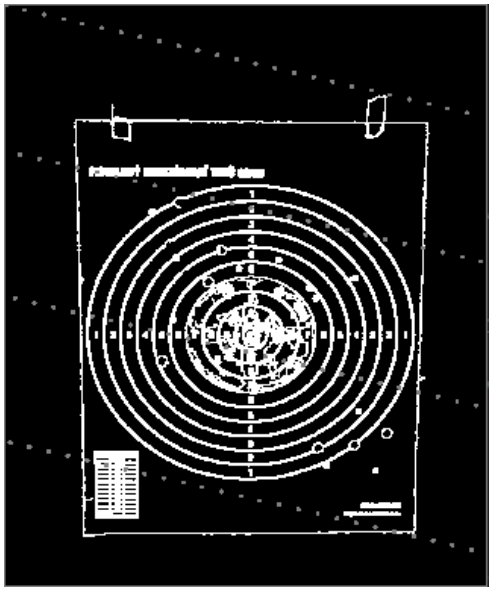

Obrázek 1: Vizualizace problému obrázků v odstínech šedi
a) vstupní obraz, b) Obraz v odstínech šedi, c) výsledek hranové analýzy nebarevného obrazu




Obrázek 2: Vizualizace postupu extrahování hran terče a okolí
a) Vstupní obraz, b) Jednotlivé S a V kanály, c) Hranové analýzy S a V kanálů d) Obraz s výslednou kombinací hranových obrazů

### 2.1.2 Indexace jednotlivých objektů v obraze

Po hranové analýze následuje indexace (označení hodnotou) jednotlivých objektů v obraze. Pro indexaci je použit algoritmus flood fill [5], který se šíří tak dlouho dokud nenarazí na hranu a označuje každý pixel celistvého objektu číslem. Pro zefektivnění algoritmu a pozdějšího zpracování co nejmenšího množství objektů se pro šíření používá jen určitá podmnožina startovních bodů v obraze, které jsou rovnoměrně rozloženy a je jich vždy stejný počet (v aplikaci je tento počet bodů nastaven na 100) viz obrázek 3 a jeho šedé startovní body. Po každém naindexovaném objektu se vezme další startovní bod a pokud již nebyl navštíven, tak je to startovní bod nového objektu, pro který potřebujeme novou hodnotu. Vizualizaci výsledku algoritmu flood fill [5] můžeme vidět na obrázku 4,kde má každá oddělená oblast vlastní barvu. Černé oblasti vyznačují oblasti, kde algoritmus nebyl použit kvůli rozložení a omezenému počtu startovních bodů.

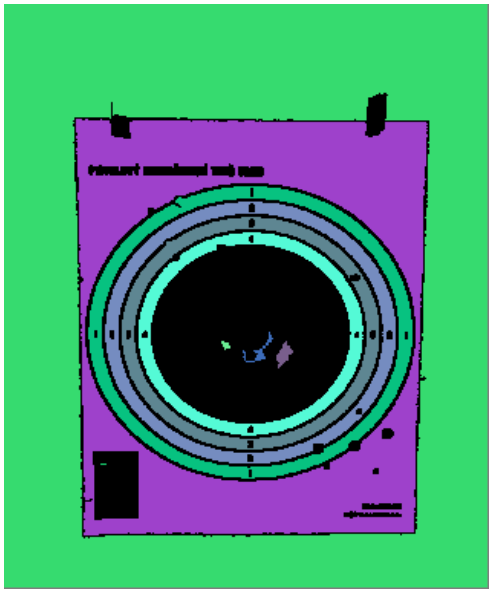


Obrázek 3: Vizualizace startovních bodů rozložených v obraze

Po indexaci objektů se výsledný naindexovaný obraz co nejvíce zmenší stejným způsobem jako byl zmenšen původní pořízený obrázek s tím rozdílem, že maximální velikost dimenze byla experimentálně stanovena na pouhých 160 pixelů. Při zmenšování bylo kvůli zachování hodnot indexů použita interpolace nejbližším sousedem.

Toto je nutné provést, jelikož při testování na mobilním telefonu celý proces extrakce terče trval přes 1 vteřinu, pokud ke zmenšování nedošlo. Toto bylo nemyslitelné s ohledem na to, že aplikace je navržena tak, aby kontinuálně pořizovala snímky z kamery mobilního telefonu. Jedná se tedy o docela dost kritickou část aplikace, která musí běžet co nejrychleji aby snímaní vypadalo pro uživatele plynule.

Zmenšení obrazu pro účel extrakce důležitých příznaků terče, které popíšu níže v této kapitole, navíc vůbec neovlivňuje přesnost transformace terče.



Obrázek 4: Vizualizace výsledku flood fill algoritmu

### 2.1.3 Extrakce příznaků z dříve naindexovaných objektů

Na naindexovaném zmenšeném obrázku dále extrahuji důležité příznaky pro určení objektu terče v obrázku, jakými jsou plocha objektu a rohy nejmenšího čtyřúhelníku, který daný objekt obklopuje. Plocha se vypočítá jednoduše jako součet všech pixelů s hodnotou indexu tohoto počítaného objektu. Pro určení zmíněných rohů se postupně pro daný objekt nachází maximální i minimální souřadnice $x$ a $y$ (celkem tedy 4 hodnoty), ze kterých se následně vytvoří jednotlivé body rohů, například pro levý horní roh se bod vytvoří z minimální x-ové a minimální y-ové souřadnice. (V OpenCV [29] je bod [0;0] položen v levém horním rohu)

Z těchto rohů dále vypočítám hodnotu, která má reflektovat kolik procent daný obklopující čtyřúhelník zabírá obrazu a jak moc je ve středu obrazu. Vypočítá se jako součet druhých mocnin x-vých a y-vých souřadnic všech čtyřech rohů od středu obrazu, který se následně normalizuje tak, že se tento součet podělí maximální možnou hodnotou, které se počítají z krajních rohů obrazu. Tato hodnota pak nakonec nabývá hodnot v intervalu <0;1> kde čím vyšší je tato hodnota tím více čtyřúhelník zabírá obrazu, a nebo je jeho těžiště více vychýlené od středu.

Jak jsem jíž nastínil, počet objektů je v obraze výrazně zredukován maximálním počtem startovacích bodů pro Flood fill [5], který je nastaven na 100. Počet objektů je tedy téměř vždy menší než 100, jelikož více startovních bodů může být umístěno ve stejném objektu. Toto dost pomáhá v časové i paměťové náročnosti při výpočtu příznaků pro konkrétní objekty v obrazu. Pokud bychom měli terč s pozadím kde by bylo mnoho hran, bez této redukce by se zde mohlo najít až 3000 objektů, z nichž by většina byla velmi malá a nepodstatná pro další výpočty.

### 2.1.4 Výběr nejvhodnějšího objektu pomocí dříve vypočtených příznaků a nalezení jeho skutečných rohů

Z vypočtených příznaků můžu tedy přistoupit k výběru nejvhodnějšího objektu, reprezentující můj hledaný terč. Pro tento krok bych mohl použít například metody strojového učení, nicméně jak jsem zjistil a otestoval, pro to, jak je aplikace navržena a jak komunikuje s uživatelem (bude popsáno v dalších kapitolách) bohatě stačí následující jednoduchý algoritmus.

Výše zmíněný normalizovaný součet vzdáleností rohů od středu nesmí překročit určitou hodnotu (tímto eliminuji situace kdy hrana terče v nějakém místě splyne s okolím, objekt tak přeteče do celého obrazu a extrahoval by se tím pádem špatný objekt.). Tato hodnota je experimentálně stanovena na 0.85. Tímto vyloučíme všechny objekty, které mají příliš velký obklopující čtyřúhelník nebo jsou příliš vychýlené od středu obrazu. Ze zbývajících objektů pak vybereme ten, který má největší plochu. Říkám tedy, že objekt terče je pravděpodobně největší objekt z objektů, které nemají příliš velký obklopující čtyřúhelník a zároveň není jejich těžiště příliš vychýleno od středu obrazu. Z testování vyplývá, že prakticky stačí aby na kameře byly vidět všechny čtyři rohy terče a pokud hrana terče někde nesplyne s okolím a tuto situaci nedokážeme eliminovat dilatací hranového obrazu, tak se objekt terče správně najde. Pokud se nenajde žádný objekt, který by splňoval tato kritéria, aplikace zažádá o další snímek z kamery.

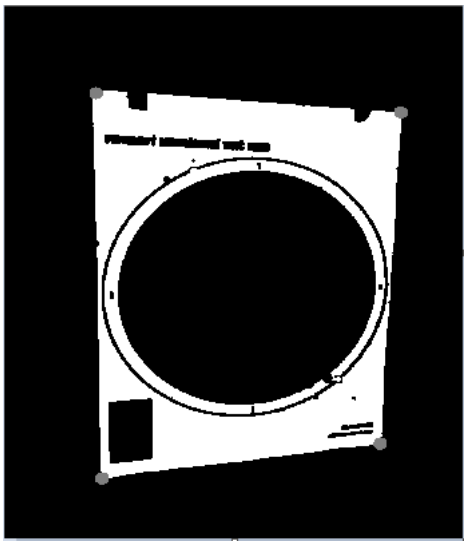
Jelikož mohou nastat určité krajní případy, kdy se najde falešný terč, potenciální objekt terče se ještě otestuje metodou, která bude popsána v další podkapitole 2.1.5.

Jelikož další zpracování už není tak časově náročné, můžu další výpočty již dělat na původně nezmenšeném obrazu pro lepší přesnost transformace terče. Jediné, co tedy potřebuji je převést pozice rohů obklopujícího čtyřúhelníku v malém obraze do velkého obrazu. To udělám jednoduše tak, že zjistím kolikrát se malý obraz zmenšil v porovnání s původním obrazem a body rohů touto hodnotou přenásobím.

V dalším postupu potřebuji získat skutečné rohy objektu. To můžu udělat algoritmem který hledá pro každý roh obklopujícího čtyřúhelníku nejbližší bod, který je označen indexovou hodnotou našeho objektu představující terč. Algoritmus je založen na tom, že pro každý roh víme, jakým směrem máme skutečný roh hledat. Například pro levý horní roh hledáme bod, který je v pravém dolním kvadrantu relativně k rohu obklopujícího čtyřúhelníku. Pro zefektivnění je zbytečné hledat nejbližší bod v celém kvadrantu, jelikož pokud by nejbližší bod byl hodně vzdálen od původního rohu, objekt v žádném případě nemůže být podobný obdélníku (nebo je terč pořízen v nějaké neočekávané perspektivě) a tím pádem nesplňuje předpoklad, že terč je nakreslen vždy na papíře, který je obdélníkový (nebo čtvercový). Pokud by se tedy stalo, že by se roh nenašel, nemá smysl pokračovat a aplikace si zažádá o další snímek.

Hledá se tedy jen v určitém čtvercovém pod-kvadrantu, jehož velikost strany se rovná 20% výšky obrazu. toto má za následek také to, že čím víc jsou strany terče vychýleny z vodorovné pozice, tím je větší šance, že se terč nenajde. Tahle vlastnost nutí uživatele, aby terč nenatáčel v nějakém hodně vychýleném úhlu vzhledem k hraně kamery a tím pádem pozdější perspektivní

transformace může být mnohem přesnější. Na obrázku 5 můžeme vidět binarizovaný obraz terče s nalezenými rohy. Vidíme také, že uživateli není zakázáno pořizovat terč v různých perspektivách, jak by mohlo vyznít z dřívějšího textu, ale je zamezeno jen určitým extrémům.



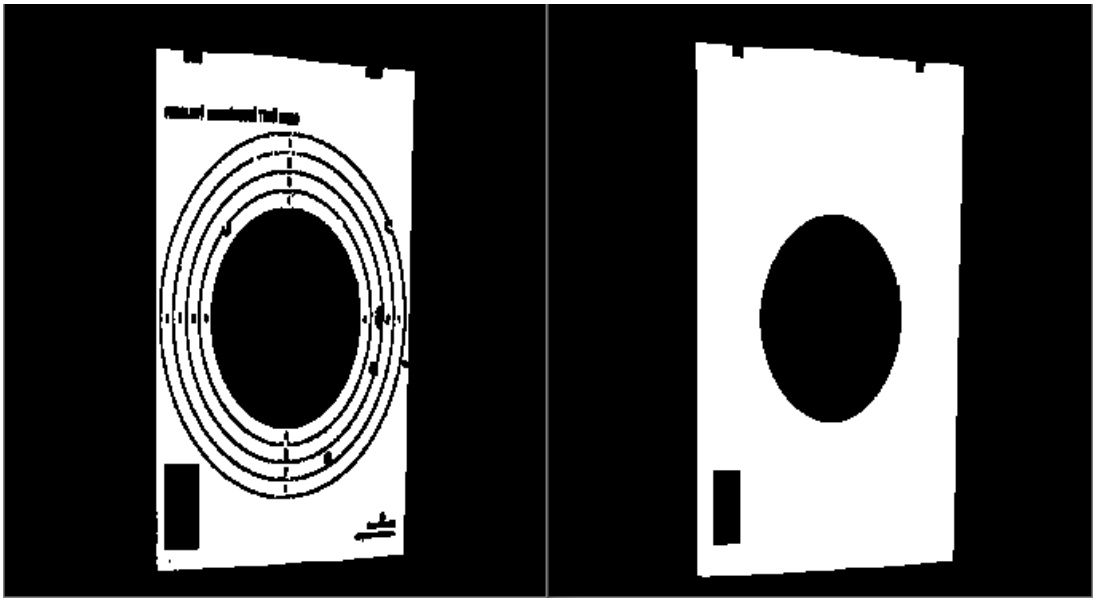
Obrázek 5: Binarizovaný obrázek s nalezenými rohy

### 2.1.5 Otestování potencionálního objektu terče

Po nalezení nejvhodnějšího objektu terče a jeho rohů, můžu přistoupit k testu, zda se jedná o terč. Tento test v podstatě zjišťuje, jestli po perspektivní transformaci daného objektu obraz obsahuje pozadí nebo ne. Pokud pozadí neobsahuje (nebo jen velmi málo), pak se s největší pravděpodobností jedná o objekt podobný čtyřúhelníku. Tento předpoklad platí hlavně pro čtyřúhelníkové objekty jako je například podklad terče. Jinými slovy zde testuji to, jestli pozadí nějakým způsobem nezasahuje do prostoru čtyřúhelníku tvořeným čtyřmi rohy.

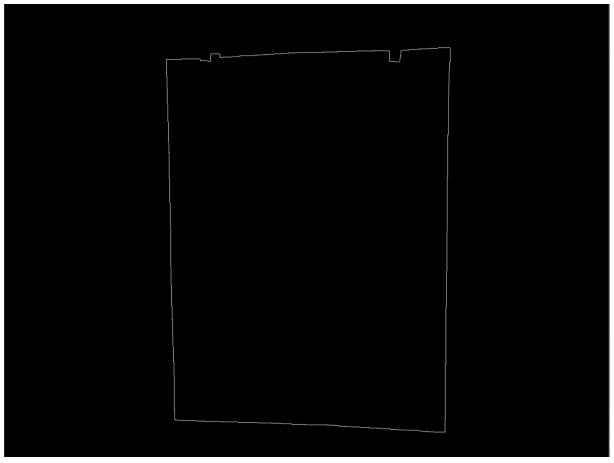
Jelikož oddělená oblast symbolizující terč většinou nepokrývá celý terč (například mezikruží v terči kvůli hranám kruhů), je nutné celý terč označit jako jeden homogenní objekt, aby se jako pozadí braly pouze objekty, nacházející se vně hranice terče. Prvním krokem je binarizovat obraz tak, aby pixely našeho objektu terče měly hodnotu 255 a zbytek měl hodnotu 0. Dále na tento binarizovaný obraz aplikuji morfologickou operaci dilatace [4], abych měl jistotu, že vedle hranice objektu terče, není žádný jiný objekt (například průstřel), který by znemožnil přesné určení hranice, které je nutné pro další zpracování. Na obrázku 6 můžeme vidět příklad dilatovaného a nedilatovaného binarizovaného obrazu, kde pravá hrana terče splývá s ostatními objekty, dilatací je tento problém vyřešen.

Po předchozích přípravných krocích můžu přejít k algoritmu určení hranice objektu [6], kde si určíme počáteční bod hranice (například levý horní roh objektu) a pokračujeme podle následujícího postupu: nacházíme-li se uvnitř objektu, otočíme se doleva a posuneme o jeden pixel. Pokud se nacházíme vně objektu otočíme se doprava a posuneme se o jeden pixel. Tento



Obrázek 6: Binarizovaný obrázek před a po dilataci

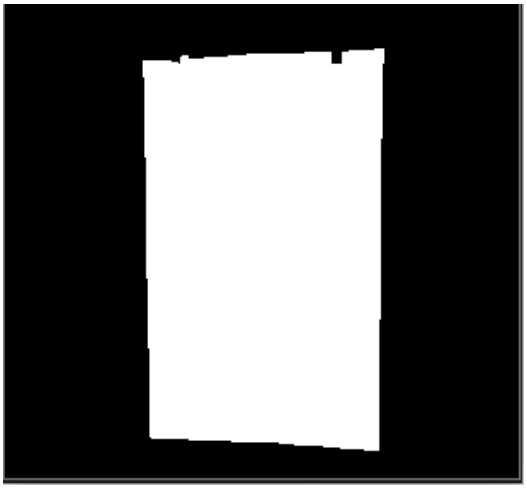
postup opakujeme do té doby, dokud nedojdeme do startovního bodu. Na obrázku 7 vidíme přesně určenou hranici, kterou využijeme v dalším kroku.



Obrázek 7: Hranice objektu

Po nalezení přesné hranice objektu provedeme na obrazu s vyobrazenou hranicí indexaci s využitím flood fill algoritmu [5] tak jak bylo popsáno výše. Výsledný obrázek 8 se tedy dělí na dva objekty oddělené hranicí. Objekt s indexem 2 by měl být terč, protože je uvnitř hranice. Objekt s indexem 1 je pozadí, protože obtéká hranici. Tímto jsem se zbavil dalších objektů uvnitř terče a můžu přistoupit k perspektivní transformaci.

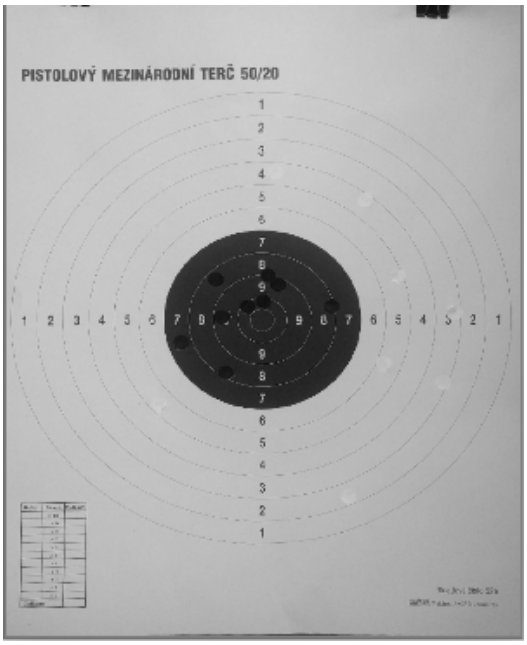
Na základě čtyřech rohů provedu perspektivní transformaci [7] do nového obrazu a spočítám kolik pixelů má index 1 (patří do pozadí obrazu). Pokud jich je více než 2% z celého obrazu,



Obrázek 8: Testovaný objekt po indexaci

Test terče je negativní a aplikace si zažádá o další snímek z kamery. V opačném případě aplikace vyhodnotí objekt jako terč.

Následně proběhne perspektivní transformace [7] tentokrát již na původním snímku. Transformace proběhne do obrazu s normalizovanou velikostí. Výsledný obraz transformovaného normalizovaného terče můžeme vidět na obrázku 9



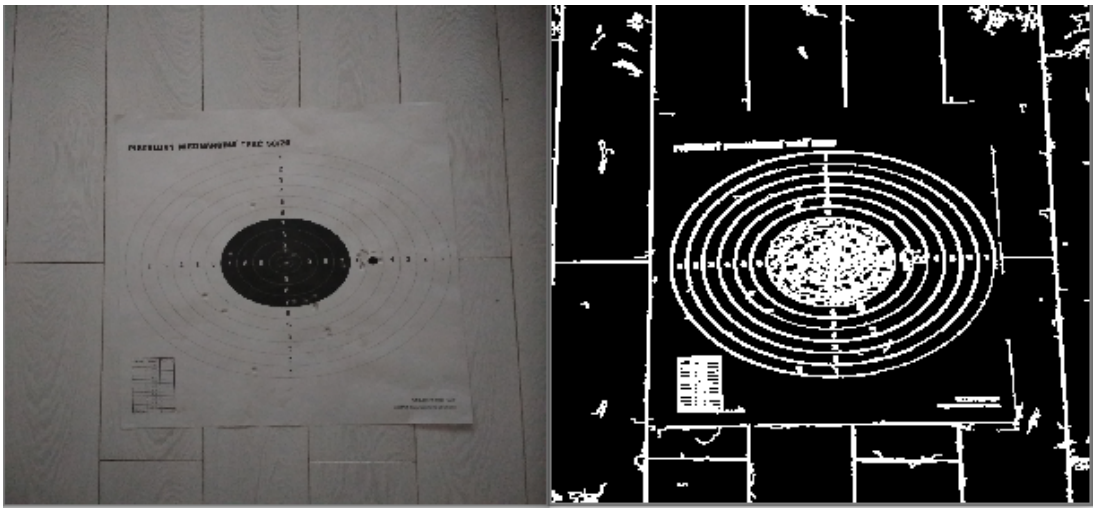

Obrázek 9: Transformovaný normalizovaný terč

## 2.2 Zhodnocení segmentace a popsání případných problémů s ní souvisejících

Tato kapitola rozebere, jaké jsou hlavní problémy řešení vysegmentování terče z obrazu, které bylo popsáno v předchozí kapitole. Budu se zde zabývat případy, kdy lidské oko jasně pozná, že na kameře terč je, ale aplikace ho z nějakého důvodu nedokáže vy-segmentovat. Další možností je, že aplikace vy-segmentuje objekt falešného terče, jelikož dříve uvedená kritéria toto ve 100 procentech případů nevylučují.

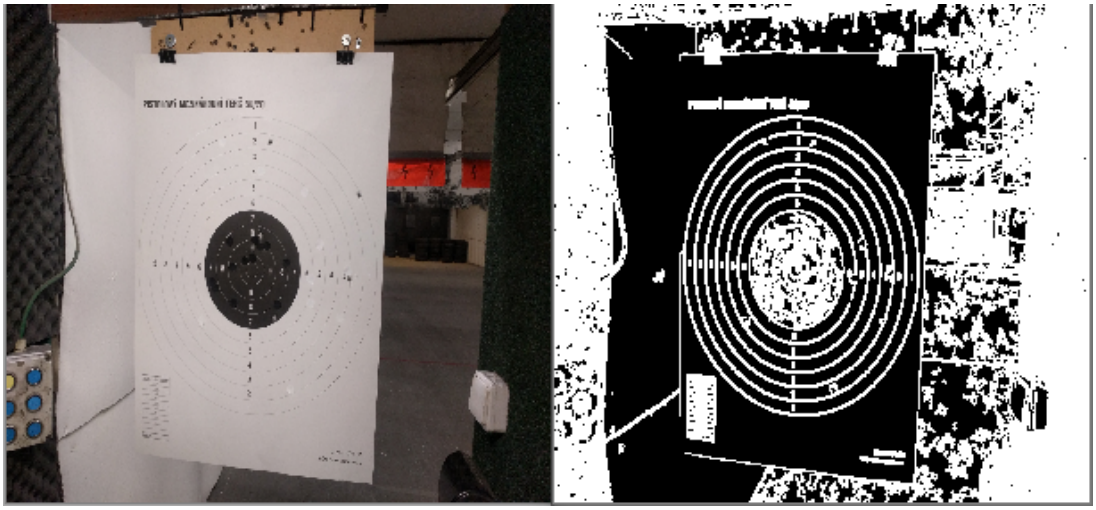
### 2.2.1 Problém nenalezení terče

Případ kdy aplikace zpracovává obraz s terčem a nedokáže ho najít je nejčastěji způsoben tím, že hranová analýza obrazu ne vždy odhalí celou hranici terče, a tak při aplikování flood fill algoritmu [5] objekt terče přeteče do pozadí obrazu a tím pádem pak při výběru vhodného objektu vůbec nebude nalezen. Tento problém nastává v situacích, kde hrana terče v nějakém místě splývá s pozadím. Spoustu těchto případů vyřeší výše zmíněná dilatace [4] hranového obrazu tak, že vyplní díry hranice. Extrémní případ nastává, když je terč pořízen na bílém pozadí v odstínu podobném podkladu terče. Tento případ na střelnicích naštěstí nastává jen velmi zřídka a když nastane, nejde s tím nic dělat a terč se musí přesunout na jiné pozadí. Příklad tohoto případu je vyobrazen na obrázku 10. Možným řešením by mohlo být, detekovat největší kružnici terče, nicméně to by se aplikace omezila jen na terče s podobným kruhovým rozložením zón zásahů a v budoucnu by nebylo tak jednoduché ji rozšířit i na jiné typy terčů.



Obrázek 10: Příklad terče na splývajícím pozadí

Dalším příkladem může být, že terč je pořízen se splývajícím pozadím jen z části (například na snímcích s terčem i bílou stěnu která je na straně), Tento problém se už ale dá řešit tím, že uživatel změní úhel kamery a terče, tak aby splývající stěna již vidět nebyla. Tento případ můžeme vidět na obrázku 11.



Obrázek 11: Příklad terče kdy částečně splývá s okolím

Dalším nepříliš pravděpodobným problémem je situace, kdy se terč nějakým způsobem rozdělí na dva objekty. Toto se u mezinárodního terče 50/20 může stát v případě, že se naleznou hrany (a rozdělí tak terč) v nejužším místě, kde je nejméně prostoru vedle největší kružnice. Tato místa jsou dvě (na obou stranách terče) a hrany musí rozdělovat terč na obou místech, aby se terč rozdělil napůl. Tato situace je teoreticky možná, když střelec na obou místech přesně prostřelí ono zúžení, což je ale velice nepravděpodobné. Na následujícím obrázku 12 je tato situace simulována.


Obrázek 12: Příklad nalezených hran v kritických místech terče

Terč nemusí být nalezen také z důvodu, že nezabírá v obrazu dost místa (aplikace odmítne jakýkoliv objekt, který zabírá méně než 30% obrazu). Tuto restrikci jsem zde dal proto, aby nebyl terč příliš malý a nebyly by vidět detaily, které by byly potřeba pro další zpracování průstřelů v terči. Tímto tedy nutím uživatele k tomu, aby terč vyplňoval velkou část obrazu.

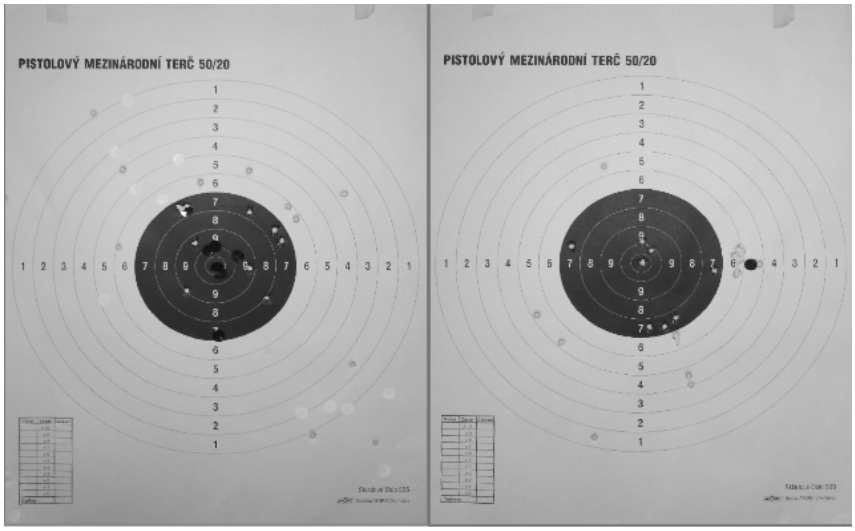
### 2.2.2 Problém nalezení nesprávného objektu

Z implementace popsané výše vyplývá, že extraktor terče, není úplně tak extraktor terče jako spíš extraktor jakéhokoliv objektu, který zaujímá v obraze dostatečně velké místo a má tvar čtyřúhelníku. Uživateli se tedy jednoduše může stát, že nebude snímat terč ale nějaký jiný objekt čtyřúhelníkový objekt, aplikace ho extrahuje. Teoreticky by se extrahovaný objekt dal nějak porovnávat s šablonou terče, nicméně mi to přišlo zbytečné, jelikož se aplikace vždy po úspěšné extrakci může uživatele zeptat, jestli to je opravdu hledaný terč nebo jestli má pokračovat v hledání. Tento stav nastane jen v případě, že uživatel nesnímá terč (nebo je terč v obraze příliš malý).

Výhodou tohoto řešení je, že bude fungovat na jakýkoliv terč, který je na čtyřúhelníkovém podkladě (Téměř všechny terče jsou tištěny na obdélníkové bílé papíry). Tato část aplikace by se také teoreticky dala znovupoužít v nějakém mobilním skeneru dokumentů.

### 2.2.3 Problém ne vždy přesné extrakce terče

S perspektivní transformací nastává menší problém v tom, že extrakce a transformace terče do jednotné podoby není vždy 100% přesná. Tato nepřesnost vzniká z důvodů pořizování terče v různých perspektivách nebo nepřesného určení rohů terče. Podklad terče může být také nějak deformován. Na obrázku 13 si můžeme všimnout, že například levá i pravá část největšího kruhu v terči je u pravého obrázku blíže okraji.




Obrázek 13: Ukázka nepřesnosti transformace terče

Problém pak může nastat v konečném porovnání s již uloženou šablonou pro daný typ terče, kde průstřely, ležící na hranicích různých bodových ohodnocení, můžou být obodovány jinak, než by ručně bodoval střelec. Uživatel může do jisté míry ovlivnit větší přesnost tím, že se bude snažit terč zachytit, tak aby jeho strany byly co nejvíce rovnoběžné se stranami kamery.

### 2.2.4 Časová náročnost extrakce terče

Testování časové náročnosti algoritmu extrakce terče jsem udělal na mobilním telefonu Xiaomi Redmi 4x [8] s procesorem Octa-core 1.40GHz s 3 GB paměti.

Jeden snímek z kamery trvá zpracovat přibližně 250 milisekund, pokud algoritmus dojde do části, kde se potenciální objekt naposled testuje. Pokud do této části nedojde (aplikace nenajde žádný potenciální objekt terče) zpracování se urychlí přibližně o 70 milisekund.

Při rychlosti kamery 30 FPS jsem tedy schopen zpracovat zhruba každý 7-8 snímek. V reálné implementaci si nechávám rezervu a zpracovávám každý 10. snímek.

Jelikož kamera plynule snímá a nijak se nezastavuje při zpracování snímků, uživatel si vůbec ničeho nevšimne.

# 3 Detekce průstřelů založená na CNN

Po vysvětlení způsobu extrakce a normalizace terče, můžu přistoupit k samotné detekci průstřelů v terči. Pro implementaci detekce jsem přistoupil k využití konvolučních neuronových sítí. Tato kapitola se zabývá celou problematikou detektoru průstřelů, založených na CNN. Zkoumá a porovnává různé varianty detektorů objektů a podrobně popisuje vybraný detektor. Na konci kapitoly je detektor zhodnocen.

Neuronová sít je trénována na datové sadě, obsahující již extrahované a normalizované terče.

## 3.1 Popis vytváření datové sady pro trénování CNN

Obecně pro použití neuronových sítí, potřebuji nějakou trénovací datovou sadu, na které bych síť mohl natrénovat. V této kapitole tedy popíšu, jak jsem tuto datovou sadu vytvářel.

### 3.1.1 Popis pořizování terčů do datové sady

Obrázky terčů v datové sadě byly pořizovány hlavně na střelnici v uzavřeném osvětleném prostoru. Vznikly na základě dvou návštěv jedné střelnice. zbývající obrázky byly pořízeny v domácím prostředí s pozadím stěny s jednotvárným pozadím. Pro trénování bylo použito přesně 46 obrázků pořízených z asi šesti terčů. s různě přibývajícími průstřely. Terč byl snímán po každém kole střelby. Těchto 46 obrázků obsahuje zhruba 533 průstřelů (některá díra v terči může být způsobena i více než jedním průstřelem).

Jelikož jsem potřeboval alespoň trochu zjistit úspěšnost sítě na nenatrénovaných datech, tak jsem tím stejným způsobem vytvořil i testovací datovou sadu, která obsahuje 27 obrázků s celkem 355 průstřely.

Pro rozumnou úspěšnost detekce v reálném prostředí na různých střelnicích při různém osvětlení, by bylo nejspíš vhodné, aby trénovací sada tvořila mnohem více průstřelů, nicméně nebylo v mých silách datovou sadu rozšířit do nějaké optimální velikosti a žádnou rozsáhlou veřejnou datovou sadu průstřelů jsem nenašel.

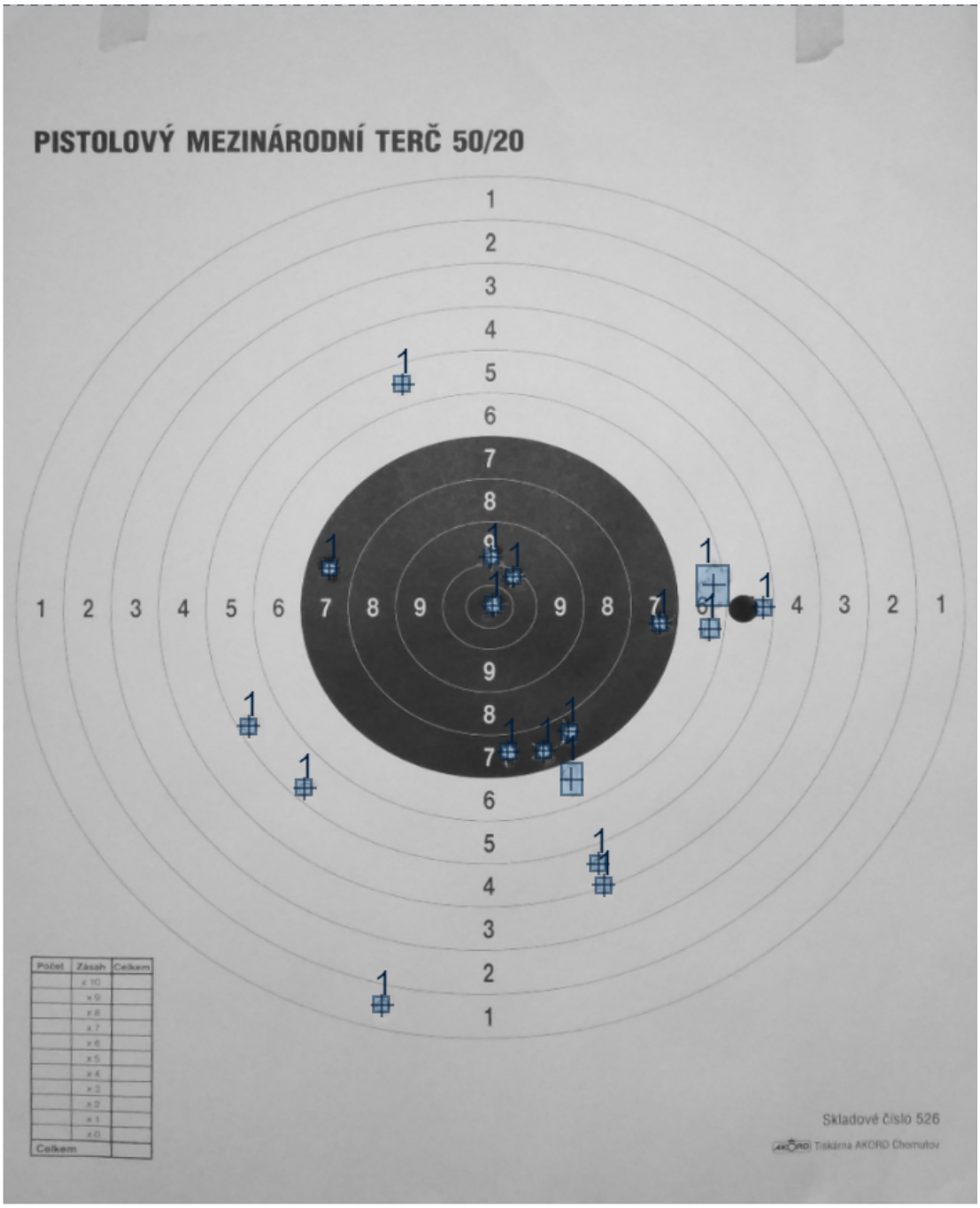
### 3.1.2 Popis označení jednotlivých průstřelů v normalizovaných terčích

Pro natrénovaní neuronové sítě jsem použil knihovnu Darknet [9], která jako vstup příjímá datovou sadu s obrázky, kde pro každý z nich existuje textový soubor popisující ohraničující rámce, pro každý průstřel v obrázku. Například pro obrázek img.jpg existuje textový soubor img.txt s následujícím obsahem:

```
0 0.716797 0.395833 0.216406 0.147222
0 0.687109 0.379167 0.255469 0.158333
0 0.420312 0.395833 0.140625 0.166667
```

První číslo na každém řádku označuje třídu, pod jakou objekt spadá (pracuji jen s jednou třídou a to je samotný průstřel, nebo více průstřelů v jedné díře), další dvě hodnoty určují střed ohraničujícího rámce a poslední dvě hodnoty značí výšku a šířku. Hodnoty jsou kódovány relativně vzhledem k velikosti obrázku a nabývají tak hodnot v intervalu <0; 1>.

Pro označení obrázků a vygenerování příslušných textových souborů jsem použil nástroj Boobs [10]. Na obrázku 14 můžeme vidět vizualizaci anotace průstřelů jednoho terče.




Obrázek 14: Vizualizace anotace průstřelů

## 3.2 Srovnání některých metod detekce objektů a jejich krátký popis fungování

V této kapitole blíže popíšu takzvané konvoluční neuronové sítě speciálně pak tzv. YOLO sítě [11]. Následně srovnám YOLO sítě s některými jinými metodami pro detekci objektů. Blíže popíšu hlavně R-CNN, fast R-CNN a faster R-CNN architektury [15] [16] [18]. Zmíním se také o možnosti extrakce HOG [23] příznaků s následnou klasifikací a o takzvané MMOD [24] metodě pro detekci objektů. Tyto dvě metody jsem zkoušel použít.

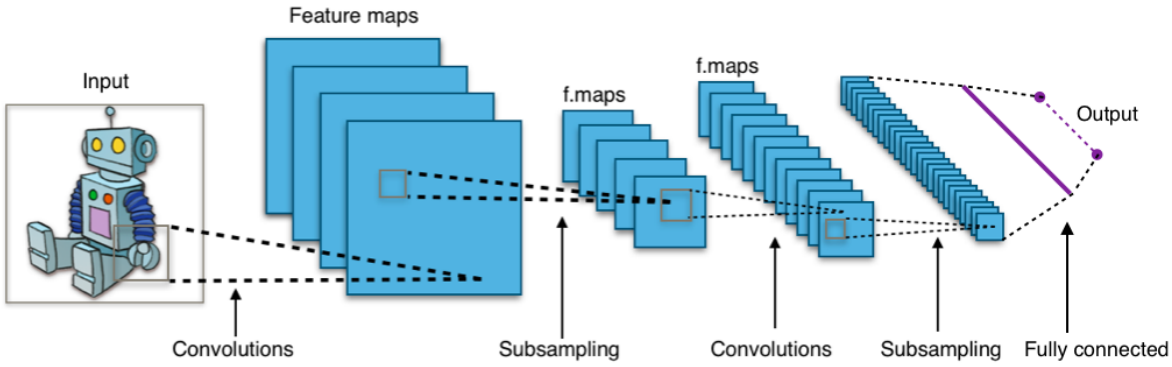
### 3.2.1 Konvoluční neuronová sít

Základem většiny detektorů, které zde budu popisovat je konvoluční neuronová síť. Je to jedna z typů hlubokých neuronových sítí, většinou použita k analýze vizuálních obrázků.

Skládá se ze vstupní vrstvy následované skrytými vrstvami. Na konci sítě se většinou používají plně propojené perceptronové vrstvy.

Na vstup sítě obvykle posíláme obrázek. Skryté sítě se pak skládají z bloků tří vrstev, kdy první vrstva je vrstva konvoluční. Tato vrstva provádí operaci konvoluce nad vstupním obrazem. Druhá vrstva může být takzvaná pooling vrstva která zmenšuje výstup konvoluční vrstvy. Další vrstva je většinou aktivační. Jeden typ z těchto aktivačních vrstev se nazývá ReLU. (viz: [12]) a je jedna z nejpopulárnějších v konvolučních neuronových sítích. Funguje tak, že nemění kladné hodnoty složek vstupní příznakové mapy a záporné přepíše na nulu.

Sít má obvykle propojeno více těchto bloků za sebou. Čím větší hloubka sítě, tím víc je celý průchod sítí náročnější na výpočet a paměť. Na obrázku 15 můžeme vidět typickou CNN.




Obrázek 15: Typická struktura konvoluční neuronové sítě - zdroj: [12]

### 3.2.2 Příklady některých detektorů používající CNN

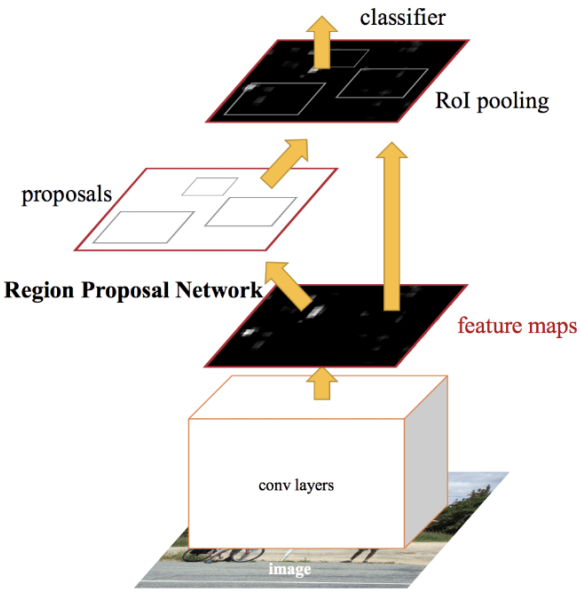
Jako příklad přístupu k detekci objektů v obraze se nabízí kombinace klasifikační neuronové sítě s posuvným okénkem, které by zkoušelo různé části obrázku v různém měřítku posílat do klasifikátoru. Tento klasifikátor (konvoluční neuronová sít) by pak jednoduše rozhodl, zda v

oblasti objekt je nebo ne. Tento proces je ale, docela výpočetně náročný, protože oblastí pro klasifikaci v obrazu je docela dost.

Přišlo se tedy s vylepšením v podobě takzvaných R-CNN (Region - convolution neural network) [15]. Jednoduše řečeno, byl vymyšlen algoritmus, jak vybrat pouze omezené množství oblastí pro následnou klasifikaci, tak aby výpočetní náročnost nebyla tak vysoká. Tyto návrhy oblastí se následně posílají do konvoluční neuronové sítě, jejímž výstupem je vektor příznaků, který se pak klasifikuje pomocí SVM [17]. Navzdory vylepšení, výpočetní náročnost byla stále dost velká na to, aby detektor běžel v reálném čase.

Další vylepšení přišlo s takzvanýma Fast R-CNN [16]. Místo toho, aby se pomocí algoritmu v R-CNN [15] navrhly oblasti ze vstupního obrázku, pomocí konvoluční neuronové sítě se vygeneruje příznaková mapa, ze které se pomocí výše zmíněného algoritmu vyberou oblasti pro následný klasifikátor.

Nakonec se Fast R-CNN [16] ještě trochu vylepšilo a místo algoritmu výběru oblastí pro klasifikátor se použila další oddělena neuronová sít pro predikce oněch důležitých oblastí. Tento systém se nazývá faster R-CNN [18] Vizualizaci celého procesu Fast R-CNN můžeme vidět na obrázku 16




Obrázek 16: Vizualizace faster R-CNN detektoru - zdroj: [18]

### 3.2.3 YOLO konvoluční neuronová sít

YOLO [11] [11] (you look only once) je metoda detekce objektů v reálném čase. Na rozdíl od většiny přístupu k detekci objektů, tato speciální konvoluční neuronová sít, určená pro klasifikaci a detekci objektů, zpracovává celý obraz jako jeden vstup. Každý obrázek projde sítí pouze jednou.

V dalším textu bude zevrubně rozebrán princip fungování YOLOv1 [11], která je základ pro všechny další verze. Na počátku se vstup rozdělí na N x N políček, kde každý z nich je zodpovědný za určení tzv. predikce. Každá buňka ze sítě políček predikuje X ohraničujících rámečků, kde každý z nich obsahuje 5 komponent $(x, y, w, h, jistota)$ - [x; y] jako bod středu, w a h je výška a šířka, jistota je v podstatě pravděpodobnost, jestli se v políčku vyskytuje nějaká z námi definovaných tříd. Koordináty rámečku jsou normalizovány do intervalu <0; 1> relativně k velikosti dimenzí celého obrazu.

Každá buňka generuje X těchto rámečků. Nakonec tedy máme N x N x X x 5 čísel na výstupu sítě.

Ke každé buňce se také vztahují pravděpodobnosti výskytu jednotlivých tříd. Pokud k původnímu vektoru rámečků přičteme pravděpodobnosti jednotlivých tříd, dostaneme N x N x (X * 5 + C) velký vektor reálných čísel na výstupu sítě.

Jakmile si uvědomíme, jak má vypadat výstup v YOLO síti, tak je jednoduché si domyslet, že sít je vlastně obyčejná konvoluční neuronová síť, která má daný nějaký formát výstupu, pomocí kterého se pak také učí. (Na základě formátu výstupu jsou tvořena i trénovací data)

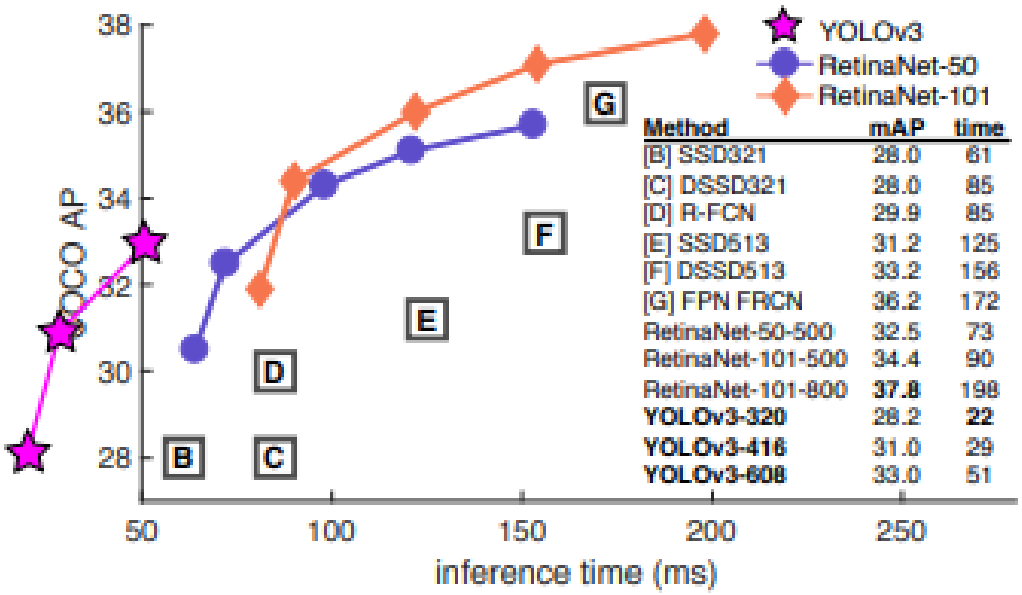
### 3.2.4 Srovnání architektury YOLO a některých ostatních přístupů k detekci objektů

Zde srovnám další vybrané architektury detektorů s různými YOLO architekturami. Srovnávat se bude časová náročnost detektorů a jejich takzvané mAP [30] s minimální hodnotou IOU [31] ohraničujících rámečků stanovenou na 0.5. Testování bylo provedeno na datové sadě COCO test-dev [19] Srovnání můžeme vidět v následujícím obrázku 17 viz. zdroj [21].

Na grafu je vidět, že YOLO architektura je jasně nejrychlejší, nicméně z daných architektur není nejpřesnější. Pro aplikaci, která bude běžet hlavně na mobilním zařízení, kde jsou omezené výpočetní prostředky, je tedy vhodné zvolit metodu, která je rychlá. V tomto ohledu je tedy architektura YOLO velice výhodná, nicméně její úspěšnost není tak vysoká jako u jiných metod v grafu. S ohledem na to, že datová sada průstřelů není podle mého názoru tak složitá, toto zase tak moc nevadí.

Před výběrem architektury YOLO, jsem ještě osobně zkoušel jiné metody, kde jednou z nich bylo řešení založeno na extrakci HOG [23] a následnou klasifikaci pomocí SVM [17] zkombinovanou s posuvným okénkem. Tato metoda byla velmi časově náročná (16 vteřin pro zpracování jednoho obrázku). Další z vyzkoušených metod byl takzvaný MMOD [24] detektor zkombinovaný s CNN, který byl implementován knihovnou DLIB [25], nicméně implementace velice závisela na využití grafické karty. Bez ní byl detektor velice pomalý (až 20 vteřin pro jeden snímek) a tím pádem byl tento způsob detekce zavržen.

Pro implementaci detektoru byla tedy nakonec zvolena architektura YOLO [11] (konkrétně nejnovější verze YOLOv3 [21]) z důvodu její rychlosti.




Obrázek 17: Srovnání architektur YOLO s ostatními vybranými architekturami - zdroj: [21]

## 3.3 Popis principu fungování metod detekce objektů YOLOv3 a porovnání s předchozími verzemi

V následujícím textu bude podrobně rozebrána použitá metoda detekce objektů YOLOv3 [21], jak se liší a jaké přináší vylepšení oproti YOLO verze 1. Poté porovnám výkon a úspěšnost základní YOLOv3 architektury s YOLOv3-tiny architekturou a jejich některými variantami.

V první řadě je třeba říct, že základní architektura sítě YOLOv3 je o něco pomalejší než YOLOv2, výměnou za větší úspěšnost detekce a schopnost dobře detekovat malé objekty. Nicméně YOLO verze 3 je pořád docela dost rychlá v porovnání s ostatními detektory.

### 3.3.1 Vylepšení architektury oproti YOLO verze 1

Základní princip fungování zůstává stejný jako u YOLOv1 popsaného v podkapitole 3.2.3. kdy jako detektor slouží jedna konvoluční neuronová síť a nepoužívá se zde žádná varianta posuvného okénka nebo nějaká metoda výběru oblastí v obraze, které by měly být nějakým způsobem klasifikovány.

Vylepšení přichází s takzvanýma „skip connections“, pomocí kterých je možné použít výstup příznakové mapy jedné vrstvy hlouběji v síti, nezávisle na vrstvách v síti, které jsou mezi těmito dvěma vrstvami (jednoduše spojíme výsledek předchozí příznakové mapy s výsledkem nějaké již dříve vypočítané mapy). Tato technika je použita také v takzvaných reziduálních sítích [26]. Pomocí této metody se vypořádáme s problémem tzv. mizejících gradientů [27].

Další problém, který se nová verze snaží řešit je detekce velmi malých objektů, jelikož čím je síť hlubší, tím více se ztrácejí detaily v obraze a tím pádem i velmi malé objekty. Jsou tedy

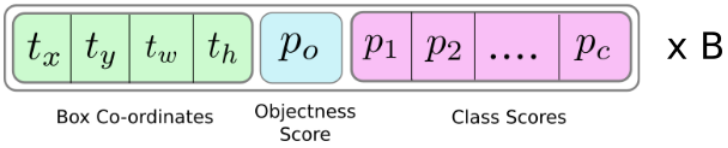
přidány vrstvy, jejichž funkce je spojit dvě nebo více dříve vypočtených příznakových map do jedné mapy. Tímto procesem můžeme zachovat informaci o malých objektech i hlouběji v síti.

### 3.3.2 Nová základní stuktura CNN použitá v YOLOv3

YOLOv3 [21] používá novou architekturu založenou na síti darknet 53, která obsahuje 53 1x1 a 3x3 konvolučních vrstev. Síť můžeme vidět na obrázku 19. K této síti je přidaných dalších 53 vrstev, to v součtu dává plně konvoluční neuronovou síť (pro redukci dimenze se zde používá stride v konvolučních neuronových vrstvách) obsahující 106 vrstev. Takto velká hloubka sítě také vysvětluje, proč je síť pomalejší než její předchozí verze.

Architektura je schopna detekovat objekty ve třech různých měřítkách na třech různých místech v síti.

V každé detekční vrstvě na třech místech v síti se aplikuje $1 \times 1$ kernel s hloubkou $B \times (5 + C)$ kde B je počet ohraničujících rámečků, které může jedna buňka ve výsledné příznakové mapě detekovat, konstanta 5 je zde pro 4 koordináty každého ohraničujícího rámečku a hodnotu jistoty každého z nich. C je pak počet tříd, které chceme detekovat. Na obrázku 18 vidíme strukturu každé buňky výsledné detekční příznakové mapy. Vizualizaci celé 107 vrstvené sítě můžeme vidět na obrázku 20.




Obrázek 18: struktura buňky ve výsledné příznakové mapě - zdroj: [21]

| | Type | Filters | Size | Output |
|---|---|---|---|---|
| | Convolutional | 32 | 3 × 3 | 256 × 256 |
| | Convolutional | 64 | 3 × 3 / 2 | 128 × 128 |
| 1× | Convolutional | 32 | 1 × 1 | |
| | Convolutional | 64 | 3 × 3 | |
| | Residual | | | 128 × 128 |
| | Convolutional | 128 | 3 × 3 / 2 | 64 × 64 |
| 2× | Convolutional | 64 | 1 × 1 | |
| | Convolutional | 128 | 3 × 3 | |
| | Residual | | | 64 × 64 |
| | Convolutional | 256 | 3 × 3 / 2 | 32 × 32 |
| 8× | Convolutional | 128 | 1 × 1 | |
| | Convolutional | 256 | 3 × 3 | |
| | Residual | | | 32 × 32 |
| | Convolutional | 512 | 3 × 3 / 2 | 16 × 16 |
| 8× | Convolutional | 256 | 1 × 1 | |
| | Convolutional | 512 | 3 × 3 | |
| | Residual | | | 16 × 16 |
| | Convolutional | 1024 | 3 × 3 / 2 | 8 × 8 |
| 4× | Convolutional | 512 | 1 × 1 | |
| | Convolutional | 1024 | 3 × 3 | |
| | Residual | | | 8 × 8 |
| | Avgpool | | Global | |
| | Connected | | 1000 | |
| | Softmax | | | |

Obrázek 19: Darknet 53 architektura - zdroj: [13]

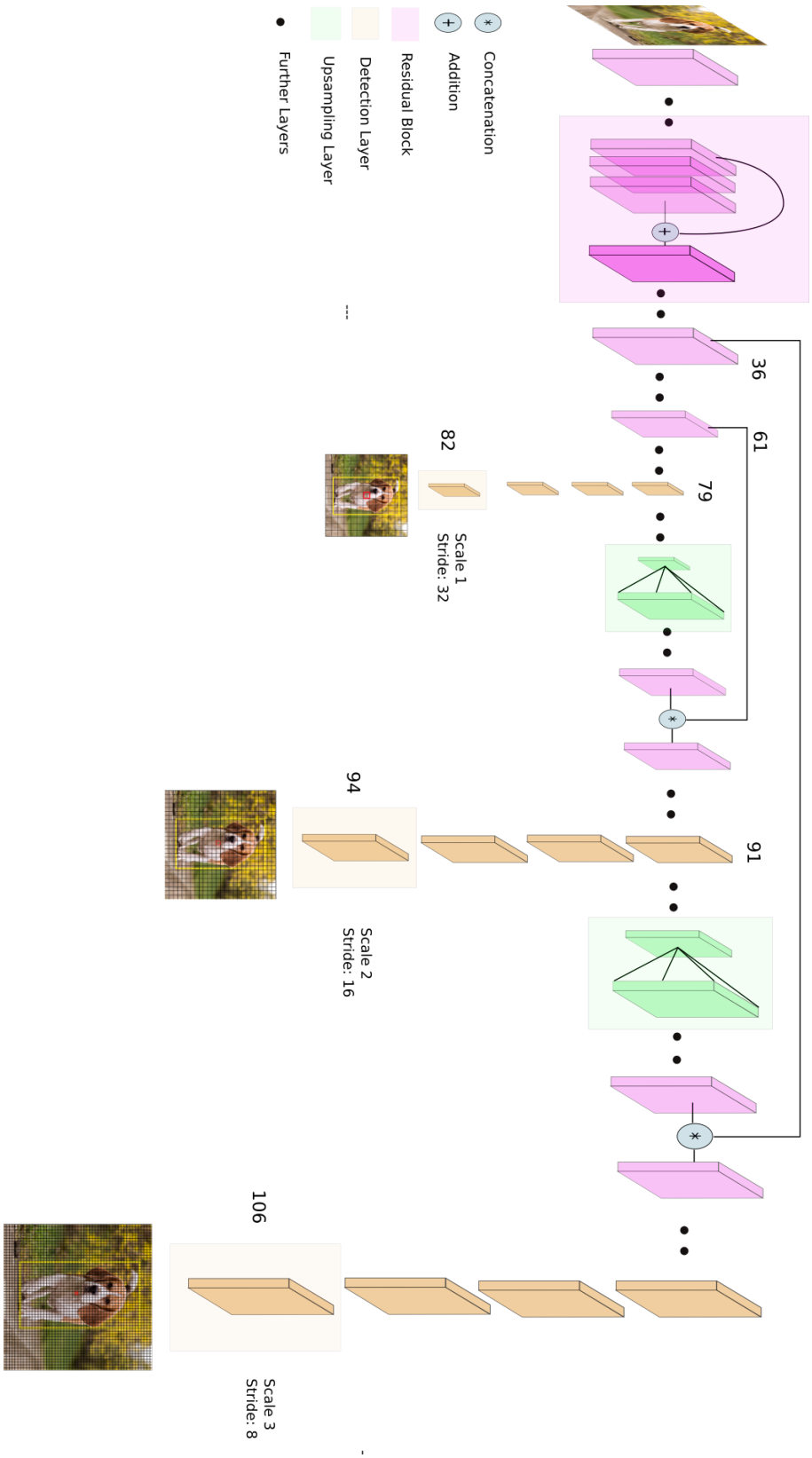
### 3.3.3 YOLOv3 tiny architektura

Mimo velmi hlubokou základní architekturu byla vytvořena i mnohem mělčí architektura, určená pro méně výkonná zařízení. Tato architektura je mnohem rychlejší, nicméně její úspěšnost není tak velká jako u základní YOLOv3 architektury.

Architektura obsahuje pouhých 24 vrstev z čehož je 13 konvolučních s velikostí 1 x 1 a 3 x 3. Na začátku sítě vstup projde sedmi konvolučními vrstvami, kde mezi nimi jsou takzvané max-pool vrstvy, které mají za úkol postupně redukovat dimenzi příznakových map.

Oproti základní síti, tato síť má schopnost pracovat pouze s dvěma měřítky objektů, jelikož jsou zde pouze dvě detekční vrstvy rozložené v síti. Před každou z těchto detekčních vrstev jsou tři konvoluční vrstvy.

Druhá detekční vrstva používá spojení dvou příznakových map, vypočtených na různých místech v síti, do jedné mapy, což nám umožní neztratit informaci o detailech v obraze, a to tím pádem vede k lepší schopnosti detekce menších objektů (toto je pro tuto aplikaci klíčové, jelikož




Obrázek 20: YOLOv3 architektura sítě - zdroj: [21]

průstřely jsou dost malé v kontextu celého obrazu, nejmenší z nich jsou přibližně 10 x10 pixelů velké). Na obrázku 21 můžeme vidět detailní popis každé vrstvy v síti, kde tzv. route vrstvy slouží právě ke spojení dvou příznakových map z různých míst v síti a YOLO vrstvy jsou ony detekční vrstvy popsané výše.

```
   layer   filters  size/strd(dil)      input                output
   0 conv     16       3 x 3/ 1    832 x1024 x   1 ->  832 x1024 x  16 0.245 BF
   1 max                2x 2/ 2    832 x1024 x  16 ->  416 x 512 x  16 0.014 BF
   2 conv     32       3 x 3/ 1    416 x 512 x  16 ->  416 x 512 x  32 1.963 BF
   3 max                2x 2/ 2    416 x 512 x  32 ->  208 x 256 x  32 0.007 BF
   4 conv     64       3 x 3/ 1    208 x 256 x  32 ->  208 x 256 x  64 1.963 BF
   5 max                2x 2/ 2    208 x 256 x  64 ->  104 x 128 x  64 0.003 BF
   6 conv    128       3 x 3/ 1    104 x 128 x  64 ->  104 x 128 x 128 1.963 BF
   7 max                2x 2/ 2    104 x 128 x 128 ->   52 x  64 x 128 0.002 BF
   8 conv    256       3 x 3/ 1     52 x  64 x 128 ->   52 x  64 x 256 1.963 BF
   9 max                2x 2/ 2     52 x  64 x 256 ->   26 x  32 x 256 0.001 BF
  10 conv    512       3 x 3/ 1     26 x  32 x 256 ->   26 x  32 x 512 1.963 BF
  11 max                2x 2/ 1     26 x  32 x 512 ->   26 x  32 x 512 0.002 BF
  12 conv   1024       3 x 3/ 1     26 x  32 x 512 ->   26 x  32 x1024 7.852 BF
  13 conv    256       1 x 1/ 1     26 x  32 x1024 ->   26 x  32 x 256 0.436 BF
  14 conv    512       3 x 3/ 1     26 x  32 x 256 ->   26 x  32 x 512 1.963 BF
  15 conv     36       1 x 1/ 1     26 x  32 x 512 ->   26 x  32 x  36 0.031 BF
  16 yolo
[yolo] params: iou loss: mse, iou_norm: 0.75, cls_norm: 1.00, scale_x_y: 1.00
  17 route  13                                     ->   26 x  32 x 256
  18 conv    128       1 x 1/ 1     26 x  32 x 256 ->   26 x  32 x 128 0.055 BF
  19 upsample                2x     26 x  32 x 128 ->   52 x  64 x 128
  20 route  19 8                                   ->   52 x  64 x 384
  21 conv    256       3 x 3/ 1     52 x  64 x 384 ->   52 x  64 x 256 5.889 BF
  22 conv     36       1 x 1/ 1     52 x  64 x 256 ->   52 x  64 x  36 0.061 BF
  23 yolo
[yolo] params: iou loss: mse, iou_norm: 0.75, cls_norm: 1.00, scale_x_y: 1.00
```

Obrázek 21: YOLOv3 tiny architektura

### 3.3.4 Popis vstupu sítě

Jako vstup se do sítě posílá extrahovaný terč v normalizované podobě, který má vždy jednotnou velikost. Jak se ukázalo velikost obrázku podstatně ovlivňuje úspěšnost sítě a dále v kapitole budou porovnány vlivy různých variant vstupních velikostí.

Jelikož barva průstřelu (ovlivňuje ji okolí za průstřelem) nijak nedefinuje průstřel, rozhodl jsem se posílat na vstup sítě terč v odstínech šedi. Toto by mělo také zlepšit požadavky na paměť a zrychlit výpočet v porovnání se zpracováním plně barevného obrázku.

### 3.3.5 Porovnání vyzkoušených YOLOv3 architektur

V následujícím textu porovnám různé varianty YOLOv3 [21]. Porovnávat budu jejich časovou náročnost a takzvané mAP [30] (mean average presicion), což je metrika, která říká jak moc je síť úspěšná na testovacích nebo trénovacích datech. Tato hodnota je číslo v intervalu <0; 1>.

| typ sítě | mAP train | mAP test | rychlost (s) |
|---|---|---|---|
| YOLOv3 darknet 53 architektura 544 x 640 | 94,5% | 85,8% | 13 |
| YOLOv3 tiny 544 x 640 | 71,9% | 59,0% | 1,2 |
| YOLOv3 tiny 832 x 1024 | 90% | 85,0% | 2,6 |
| YOLOv3 tiny 1376 x 1664 | 98,2% | 92,3% | 6 |

Tabulka 1: Porovnání jednotlivých variant YOLOv3 sítě

mAP je počítáno při IOU [31] prahu 0.5. tato hodnota říká, že detekci považujeme za pravdivou, pokud poměr obsahu průniku trénovacího boxu s výstupním boxem sítě s obsahem sjednocení těchto dvou boxů je >= 0,5. V tabulce 1 můžeme vidět všechny vyzkoušené typy sítí, jejich mAP a rychlost.

Jako první jsem zkoušel učit základní YOLOv3 [21] architekturu zobrazenou na obrázku 20. Výsledná úspěšnost byla asi 94,5% na trénovací sadě při jednotné velikosti vstupu 544 x 640 pixelů. Na testovací datové sadě byla úspěšnost 85,8%. Po měření výkonu sítě, kterou jsem používal skrz implementaci v knihovně OpenCV[29] na mobilním telefonu, jsem zjistil, že jeden obrázek touto sítí trvá zpracovat přibližně 13 vteřin. Tento čas není úplně ideální a uživatelsky přívětivý.

S úmyslem vyřešit problém nepříliš rychlého zpracování základní YOLOv3 sítě [21] se jako další varianta řešení jevila výše zmíněná tiny YOLOv3 architektura obr. 21, která se skládá z pouhých 24 vrstev. Po natrénování sítě byla úspěšnost na trénovací množině pouhých 71,9%, na testovací sadě pak 59,0%. Výsledný čas zpracování jednoho obrázku na mobilu byl průměrně 1,2 vteřiny. Čas zpracování obrázku by již byl použitelný a uživatelsky přívětivý, nicméně úspěšnost sítě rapidně klesla.

Abych našel kompromis mezi časem zpracování a úspěšností sítě, musel jsem přijít s další variantou, jak síť modifikovat. Zjistil jsem, že úspěšnost sítě dost závisí na velikosti vstupního obrázku. Zvětšovat vstup v původní hluboké síti sice ještě o něco úspěšnost zvýšilo, ale čas pro zpracování jednoho obrázku byl ještě mnohem horší, jak se dá očekávat. Zkoušel jsem tedy zvětšovat vstup v YOLOv3 tiny [22] síti. Jako další variantu vstupu jsem tedy zvolil velikost 832 $\times$ 1024. Úspěšnost sítě po naučení na trénovacích datech byla 90.0%, na testovacích pak 85.0%. Čas zpracování byl 2,6 vteřiny.

Jako poslední jsem zkusil velikost 1376 x 1664 kde na trénovacích datech byla úspěšnost 98,2% a na testovacích 92,3%. Pro zpracování potřebovala síť asi 6 vteřin. Toto je nejpřijatelnější výsledek s ohledem na úspěšnost a čas, kterého jsem dosáhnul s použitím architektury YOLOv3 tiny [22].

## 3.4 Zhodnocení vybrané YOLOv3 architektury

V této kapitole popíšu proces a parametry trénování vybrané YOLOv3 tiny [22] architektury sítě, dále podrobně zhodnotím její úspěšnost a ukážu konkrétní příklady detekce na nenatrénovaných snímcích.

### 3.4.1 Proces a parametry trénování sítě

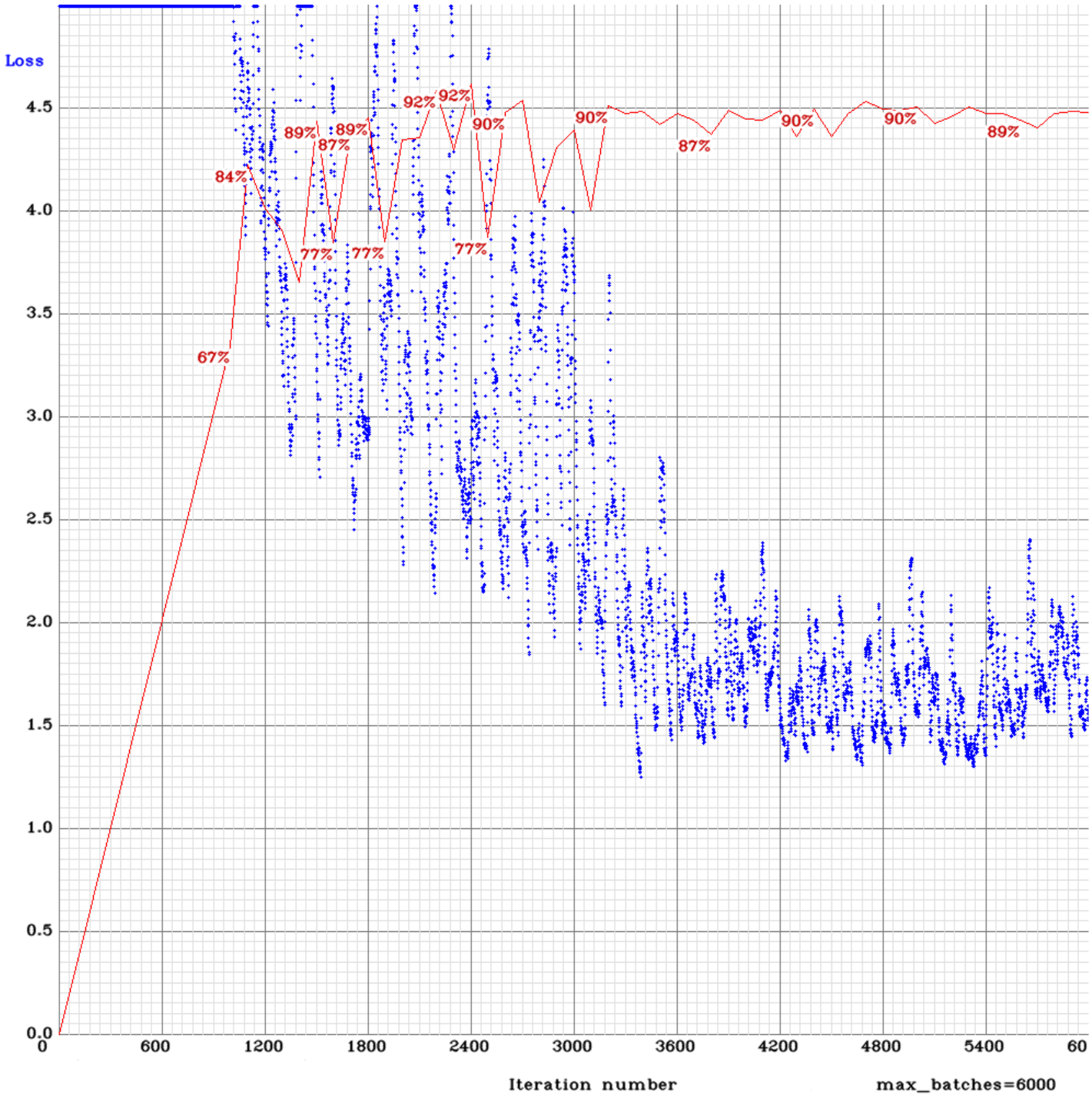
Síť byla trénována na grafické kartě GTX 1060 s 6 GB grafické paměti. Jelikož do sítě posílám velké obrázky (1376 x 1664) musel jsem nastavit hodnotu takzvané mini dávky, která říká kolik obrázků můžu při trénování poslat najednou na grafickou kartu. Tuto hodnotu jsem experimentálně nastavil na 8 jelikož větší hodnoty způsobovaly vyčerpání grafické paměti.

Dalším parametrem je velikost jedné dávky obrázků pro jednu epochu učení, který je nastaven na 64, to znamená, že každá epocha trénování obsahuje 64 snímků z trénovací datové sady. celkový počet epoch je nastaven na 6000. Čas trénování sítě na zmíněné grafické kartě pro 6000 epoch byl celkem 31 hodin.

Důležitým parametrem trénování sítě je takzvaný learning rate, který specifikuje s jak velkým krokem se můžou při trénování měnit jednotlivé váhy. Tato hodnota byla při trénování nastavena na 0,001 a ovlivňuje rychlost natrénování sítě a do určité míry i její přesnost.

Při procesu trénování jsem zároveň sledoval, jak se mění mAP [30] na testovacích datech. Nakonec jsem vybral z celého trénování právě váhy, které měly největší mAP a tedy největší úspěšnost na testovacích datech. Tímto jsem také zamezil přílišnému přeučení sítě, která by pak dobře fungovala jen na trénovacích datech. Na obrázku 22 vidíme graf zobrazující celkový postup trénování sítě kde červená křivka říká jaké je v daném bodě trénování mAP na testovacích datech.

Z grafu můžeme vidět, že trénování sítě neprobíhá úplně stabilně a chyba sítě na trénovacích datech docela dost kolísá, nicméně nakonec síť byla schopna naučit trénovací dataset s 98,2 procentním mAP.




Obrázek 22: Graf trénování sítě

### 3.4.2 Zhodnocení úspěšnosti

Jak již bylo řečeno úspěšnost sítě na trénovací sadě je 98,2% a na testovací 92,3%. Důležitými jsou pak hodnoty TP, FP a FN (true positives, false positives a false negatives).

Hodnota TP říká, kolik průstřelů bylo úspěšně detekováno z celkového počtu ručně označených „pravdivých“ průstřelů. FP je hodnota určující počet falešných detekcí, které by vůbec neměly být detekovány jako průstřel. FN říká kolik průstřelů nebylo detekováno, ačkoliv detekovány být měly. Všechny tyto hodnoty jsou zobrazené v tabulce 2 pro testovací i trénovací datovou sadu.

| **Měřená hodnota** | **Trénovací datová sada** | **Testovací datová sada** |
|---|---|---|
| **Počet skutečných průstřelů** | 533 | 355 |
| **TP** | 511 | 322 |
| **FP** | 30 | 29 |
| **FN** | 22 | 33 |
| **mAP (%)** | 98,2 | 92,3 |

Tabulka 2: Hodnoty úspěšnosti sítě

Je důležité podotknout, že výše zmíněné hodnoty jsou měřeny na surovém výstupu sítě. Síť totiž detekuje více ohraničujících rámečků kolem jednoho průstřelu, z nichž některé nebudou splňovat minimální možné IOU [31] s ručně anotovaným rámečkem a tím pádem se počítají do falešně pozitivních nálezů (tyto nálezy by měly mít sítí určenou menší pravděpodobnost toho, že se jedná o průstřel, než ostatní nálezy v okolí s větším IOU [31]). Tento problém řeším algoritmem Non-maximum Suppression (NMS) [32], který u překrývajících se nálezů, vybírá nález s největší určenou pravděpodobností. Pro tento algoritmus je třeba určit co to znamená, že se nálezy překrývají. Toto se dělá pomocí maximální hodnoty IOU [31]. Pokud dva rámce mají tuto hodnotu nižší než zmíněné maximum, pak se nálezy berou jako dva objekty ležící těsně u sebe, v opačném případě se jako správný nález považuje ten s větší pravděpodobností detekce. Tuto hodnotu mám nastavenou na 0,2 (překrytí toleruji asi z dvaceti %), jelikož v trénovací datové sadě existují rámečky, které se překrývají.

Z tohoto vyplývá, že po aplikování algoritmu NMS [32] se mAP zvýší. V mém případě se na trénovací datové sadě zvýšilo na 99,4 % a na testovací na 97,0 %.

### 3.4.3 Rozbor falešně negativních nálezů

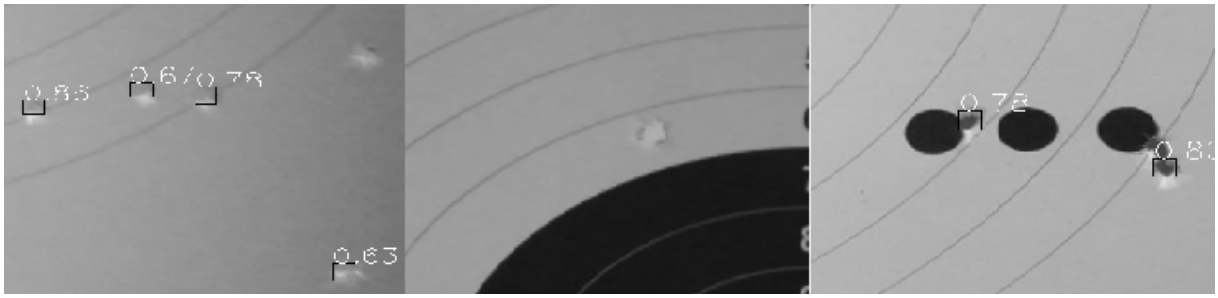
Zde rozeberu na konkrétních příkladech, kdy síť selže a nedokáže detekovat průstřel v terči. Většinou se jedná o nějakým způsobem atypické průstřely, kterých je v trénovací datové sadě velmi omezené množství. Jedná se hlavně o nepříliš výrazné světlé průstřely splývající se samotným terčem. Dalším případem jsou průstřely, které se v nějakém místě překrývají. Na obrázku 23 můžeme vidět konkrétní průstřely která síť nedokázala detekovat. V prvním a druhém výřezu se

jedná o případ nepříliš výrazných průstřelů. Poslední výřez je pak příklad dvou překrývajících se průstřelů, kde se detekuje pouze jeden z nich.

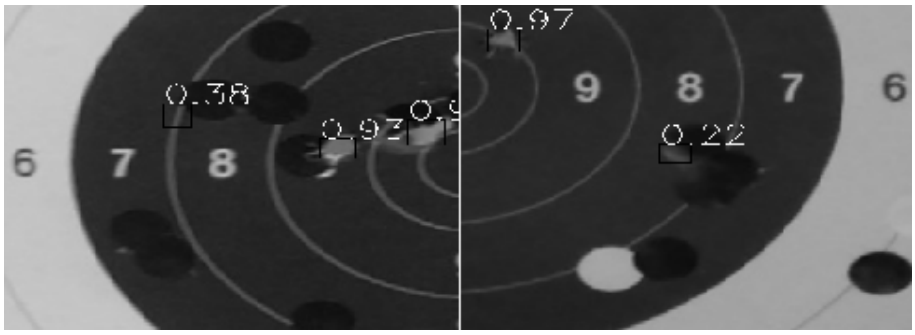
### 3.4.4 Rozbor falešně pozitivních nálezů

Opačným případem pak je případ, kdy síť detekuje hledaný objekt tam kde reálně není. Tyto nálezy se nacházejí hlavně v oblastech černých zálepek. Tato místa jsou riziková hlavně když se od nich z části odráží světlo a tvoří se na nich různé odlesky. Jelikož v trénovací datové sadě moc takových zálepek s odlesky není, síť na tyto případy není připravená. Velmi zřídka se stává, že síť nalezne falešný průstřel na obvodu soustředných kruhů, nicméně na takový případ jsem narazil pouze jednou v celé testovací a zároveň trénovací datové sadě. Na obrázku 24 můžeme vidět konkrétní případy, které síť detekovala špatně.

Z obrázků můžeme vidět, že falešně pozitivní nálezy mají docela malou jistotu, že nalezené objekty jsou skutečně průstřely (0,38 a 0,22), tím pádem, bych mohl tyto objekty filtrovat na základě této jistoty. Bohužel některé průstřely, které jsou detekovány správně mají také malou jistotu (takových průstřelů je docela málo), tím pádem bych sice nemusel detekovat nějaké falešné případy, ale také bych nedetekoval skutečné průstřely, u kterých si síť není moc „jistá“. Z důvodu malé jistoty u některých skutečných průstřelů jsem nakonec prahovou hodnotu jistoty stanovil na **0,1**.




Obrázek 23: Příklady falešně negativních nálezů




Obrázek 24: Příklady falešně pozitivních nálezů

# 4 Popis způsobu hodnocení střelce na základě jednotné šablony s bodováním pro daný terč

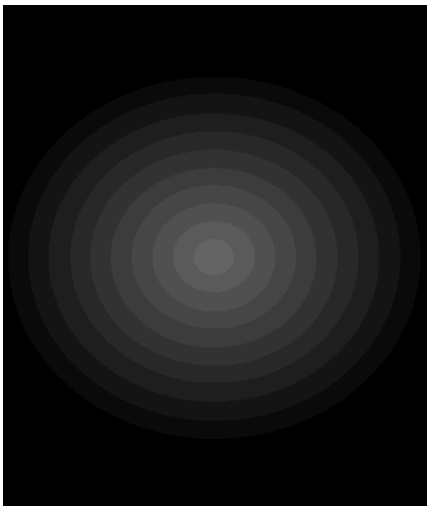
Po procesu nalezení průstřelů, jsem potřeboval vymyslet systém samotného bodování střelce na základě pozice jednotlivých průstřelů v terči. Tím že jsem terč již extrahoval a transformoval do jednotného tvaru a velikosti, nemusím řešit pozice průstřelů v terči v rámci celého snímku z kamery, ale stačí mi pouze znát absolutní pozici v rámci terče. V terči obecně platí, že různé oblasti terče jsou za více či méně bodů, tyto oblasti je tedy třeba nějak definovat.

## 4.1 Srovnání dvou možných způsobů definování oblastí bodování

Prvním nabízejícím se řešením pro mezinárodním pistolový terč 50/20 bylo detekovat všechny kruhy v terči, které oddělují jejich jednotlivé bodovací zóny. Na základě umístění těchto kruhů v terči bych pak jednoduše pro každý průstřel určil, v jakém mezikruží se nachází. V konkrétním případě mezinárodního pistolového terče 50/20 bych dokonce mohl jednoduše počítat vzdálenost průstřelu od středu, za předpokladu že bych při perspektivní transformaci zachoval poměr stran skutečného terče. Řešení závislé na detekci kruhů v terči by bylo docela přesné, jelikož by zde odpadla chybovost vzniklá perspektivní transformací terče, nicméně je zde jedna velká nevýhoda týkající se rozšiřitelnosti aplikace. Tento algoritmus detekce důležitých oblastí bodování by se totiž v každém novém typu terče lišil, a navíc by nemusel být tak jednoduchý a přímočarý jako v případě pistolového terče 50/20, kde mi stačí detekovat jen soustředné kružnice v terči. Příklad takového komplexnějšího terče můžeme vidět na obrázku 25

Problém rozšiřitelnosti aplikace na více typů terčů jsem řešil následujícím postupem. Pro každý terč vytvořím speciální šablonu, kde hodnota každého pixelu v ní odpovídá počtu bodů za průstřel v daném místě. Tato šablona má stejnou velikost jako terč po perspektivní transformaci a po „přiložení“ transformovaného terče na šablonu by si jednotlivé bodované oblasti měly v ideálním případě odpovídat. V reálném případě si oblasti neodpovídají úplně, jelikož zde vzniká chyba při perspektivní transformaci terče. Čím víc má terč na snímku přirozený obdélníkový tvar, tím víc by transformace měla být přesnější. Chyba se pak nejvíce projevuje na okrajích bodovacích zón. V porovnání s předchozím popsaným způsobem bodování není tato metoda tak přesná, ale umožňuje jednoduše rozšířit aplikaci pro více terčů v budoucnu, a proto jsem tuto metodu použil. Na obrázku 26 můžeme vidět vizualizaci oné šablony pro mezinárodním pistolový terč 50/20, kde světlejší oblasti jsou za více bodů než tmavší. Body za každý detekovaný průstřel se pak jednoduše sečtou a zobrazí uživateli.


Obrázek 25: Příklad komplexnějšího policejního parkour terče - zdroj: [39]



Obrázek 26: Vizualizace šablony pro mezinárodní pistolový terč 50/20

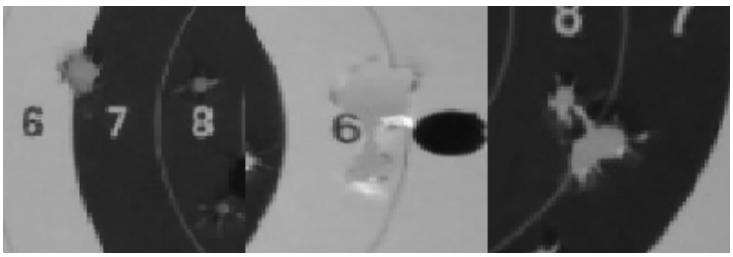
## 4.2 Proces bodování konkrétního průstřelu nebo vícenásobných průstřelů

Jak jsem již popsal výše, výstup detektoru průstřelů jsou informace o ohraničujícím rámečku pro každý detekovaný průstřel. Znám tedy přibližný střed každého průstřelu. Pro každý bod středu průstřelů se tedy jednoduše podívám na určené místo v šabloně a zjistím hodnotu pixelu. Tato hodnota říká, za kolik bodů by daný průstřel měl být hodnocen.

Jelikož v trénovací datové sadě (i v reálném prostředí) existují vícenásobné průstřely, tvořící jednu větší díru v terči, měly by tyto průstřely být hodnoceny na základě toho, kolik jich je

v jedné velké díře. Jak ale tuto informaci zjistit? Kdybych měl dostatek snímků těchto děr v trénovací datové sadě a věděl bych kolik průstřelů je v jaké díře, mohl bych v rámci detekce zavést i klasifikaci která by říkala z kolika průstřelů díra vznikla. Bohužel už základní trénovací datová sada je celkem malá a mnohonásobných průstřelů v ní moc není. Navíc je zde problém, že ani lidské oko někdy jednoduše nedokáže říct kolik průstřelů v díře skutečně je. Zavedl jsem tedy velice jednoduché řešení, které závisí na hodnotě podílu plochy ohraničujícího rámečku vztažené k velkosti obrázku. Zjistil jsem průměrnou hodnotu tohoto podílu pro jeden průstřel. Jednoduše pro každý ohraničující rámeček počítám kolikrát je jeho plocha větší než tato průměrná hodnota. Výsledek tohoto výpočtu pak zaokrouhlím a mám tím pádem informaci o tom kolik by mělo zhruba být v díře průstřelů.

Je potřeba říct, že tento jednoduchý postup může být velice nepřesný (hlavně pro díry způsobené 3 a více průstřely) a měl by uživateli sloužit pouze jako orientační informace. Na obrázku 27 můžeme vidět příklady takových vícenásobných průstřelů. Pro znázornění možné nepřesnosti postupu můžeme například vidět velmi velkou díru v prostřední snímku kde bych řekl, že jsou nejspíš 3 - 4 průstřely, nicméně tento algoritmus rozhodne, že v díře je 5 průstřelů. Obecně platí, že čím větší díra v terči, tím nepřesněji postup funguje.




Obrázek 27: Příklady vícenásobných průstřelů

# 5 Použité technologie pro implementaci detektoru

V této kapitole popíšu jednotlivé použité technologie a knihovny vedoucí k realizaci řešení, u některých zdůvodním jejich výběr.

## 5.1 Proč zrovna Android?

Hlavní důvod, proč byl pro aplikaci zvolen Android [33] je jeho široká uživatelská základna. Tento operační systém funguje převážně na dnešních mobilních telefonech a má ze všech ostatních dostupných operačních systémů největší základnu uživatelů. Také zde přispívá fakt, že jako vývojář mám k němu nejblíže ze všech mobilních operačních systémů a s vyvíjením Android aplikací jsem se již dříve setkal. I přes tyhle fakta je aplikace navržena tak, aby byla jednoduše předělatelná i pro funkci na jiných současných mobilních operačních systémech na trhu (například IOS), jak bude v dalším textu ukázáno.

## 5.2 Programovací jazyk Kotlin

Celé Android uživatelské rozhraní je napsáno pomocí programovacího jazyka Kotlin [34]. Jedná se o objektový, staticky typovaný jazyk běžící nad JVM [36] obsahující typovou inferenci (dokáže automaticky určit datový typ nějakého výrazu v jazyce). Jazyk má svoji vlastní syntaxi, nicméně je navržen pro interoperabilitu s knihovnami Javy [35], které může jednoduše volat, dokonce je na některých závislý a je s Javou tím pádem kompatibilní. Můžu ho tedy použít pro psaní Android aplikací stejně jako Javu.

## 5.3 Programovací jazyk C++

Celá logika detektoru průstřelů a bodování je napsána v programovacím jazyce C++. Tento jazyk jsem zvolil, jelikož potřebuji, aby se klíčová logika detektoru prováděla co možná nejrychleji. Pro jazyk jsou také k dispozici knihovny implementující techniky počítačového vidění a zpracování obrazu, jako je například OpenCV [29], které používám. Implementaci detektoru pak z uživatelského rozhraní volám pomocí sady nástrojů Android NDK [37], která umožňuje používat C++ kód v Android aplikaci. Tímto se také řeší jednoduchá přenositelnost celé logiky detektoru na jiný mobilní operační systém, tím že jí stačí napsat jednou v C++ a na ostatních systémech jí stačí pomocí nějakého specifického propojení používat.

## 5.4 Použití OpenCV

Obraz zpracovávám výhradně pomocí knihovny OpenCV. V této knihovně je také možnost použít implementaci YOLO konvolučních neuronových sítí [11], kterou také používám. Knihovnu používám i na straně Kotlinu, kde slouží výhradně pro jednodušší komunikaci mezi Uživatelským rozhraním a C++ logikou detektoru (Není zde napsána nějaká složitější logika, týkající se detektoru). Příklad může být třeba posílání OpenCV matice snímku z kamery do detektoru.

Nemusím tím pádem nějak řešit převod různých datových struktur na rozhraní Kotlin / C++, jelikož na obou stranách používám kompatibilní datové struktury z OpenCV. Tuto knihovnu jsem použil, protože je docela známá a léty dost prověřená, tím pádem nabízí implementace spousty technik analýzy obrazu včetně implementace konvolučních neuronových sítí.

## 5.5 NDK a JNI

NDK [37] je soubor nástrojů, umožňující použití C nebo C++ kódu v Android aplikaci. Pomocí takzvaného JNI, které definuje, jakým způsobem používat C++ knihovny z prostředí Javy [35] nebo Kotlinu [34]. Tyto dva nástroje jsou stěžejní pro mou integraci detektoru do Android aplikace a přímo se nabízí pro použití.

# 6 Popis architektury aplikace spolu s uživatelským rozhraním

Nakonec představím komunikaci jednotlivých komponent uvnitř aplikace a celkově její architekturu. Obsahem bude také popis jednoduchého uživatelského rozhraní, umožňující aplikaci testovat v reálném prostředí.

Jak už bylo řečeno architektura se skládá ze dvou částí, které se dále dělí. Tyto hlavní části pak jsou: Část určená pro komunikaci s uživatelem (napsaná v Kotlinu [34]) a logické části, která řeší veškerou logiku detektoru a rozdělování bodů. Tyto dvě komponenty jsou propojeny přes JNI [38] rozhraní. Vizualizaci celé architektury můžeme vidět na obrázku 30.

## 6.1 Popis JNI rozhraní

JNI [38] rozhraní celkově obsahuje čtyři metody mající následující funkce: Inicializace objektu detektoru, detekce terče v obrázku, detekce průstřelů a následné obodování v rámci transformovaného terče, „zničení“ objektu detektoru (když už není potřeba).

Objekt detektoru inicializujeme ihned po spuštění android aktivity určené pro poskytnutí funkcionality automatického bodování uživateli. Při vícenásobném použití metod detektoru tím pádem nemusím detektor inicializovat znova, což ušetří spoustu času. Inicializace si na vstupu žádá tři argumenty:

- Bytové pole obsahující informace o struktuře a konfiguraci YOLOv3 [22] neuronové sítě
- Bytové pole obsahující informace o naučených váhách uvnitř sítě
- Adresa OpenCV [29] matice obsahující šablonu terče pro bodování průstřelů.

Soubory s informací o neuronové síti čtu z takzvané Android asset složky na straně Kotlinu a pak je prostřednictvím Bytových polí posílám detektoru při inicializaci. Dělám to hlavně z důvodu, abych zachoval nezávislost C++ kódu na souborovém systému mobilního operačního systému.

Další metodou rozhraní je detekce terče v obrazu. Tato metoda požaduje na vstupu adresu OpenCV matice s obrázkem, na kterém by měl být terč (snímek z kamery). Po zpracování pak může vrátit adresu matice obrazu výsledného transformovaného terče. Pokud se terč nenašel vrátí se nulová adresa.

Předposlední metodou je samotná detekce a bodování průstřelů. Vstupní parametr metody je adresa matice obrazu transformovaného terče. Metoda vrací adresu matice obrazu terče s označenými průstřely a součet vypočítaných bodů pro jednotlivé průstřely.

Poslední metoda rozhraní pouze uvolňuje paměť zdrojů, se kterými objekt detektoru pracoval (soubory popisující nerunovou síť, matici obrazu šablony pro bodování a další).

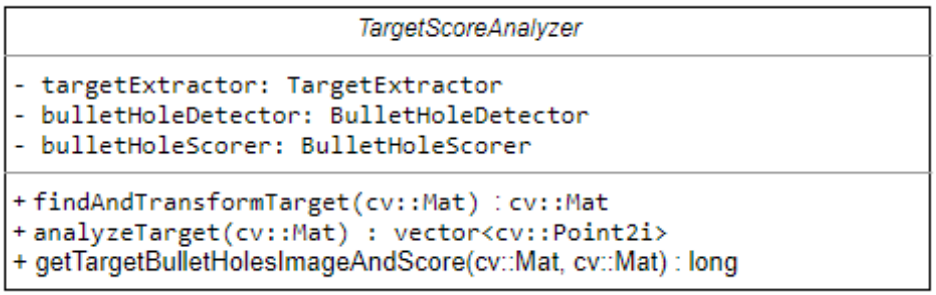
## 6.2 Popis architektury logiky bodovacího systému

Architektura logiky bodovacího systému se skládá ze tří komponent (tříd v C++). Tyto tři komponenty jsou zaštítěny pod hlavní C++ třídou, která poskytuje základní rozhraní viz. obrázek 28 popisující, jak se systémem pracovat a řídí vznik a zánik těchto tří základních objektů, kterými jsou:

- **Extraktor terče** - Stará se o nalezení objektu terče v obraze a následnou transformaci terče do jednotné podoby a velikosti tak jak bylo popsáno v druhé kapitole 2. Extraktor je připravený detekovat pouze terče s čtyřúhelníkovým podkladem (transformace na základě čtyřech rohů terče). Jsou zde definované konstanty související s analýzou objektu terče a extrakcí jako třeba prahy pro Cannyho detektor hran, nebo velikost dilatačních matic. Výstup této komponenty je transformovaný terč, nebo informace o tom, že se terč v obraze nenašel.

- **Detektor průstřelů** - V komponentě detektoru průstřelů je napsaná veškerá logika týkající se použití tiny YOLOv3 [22] konvoluční neuronové sítě pro detekci průstřelů. Komponenta pracuje s již transformovaným terčem v odstínech šedi. Komponenta má za úkol vytvořit objekt neuronové sítě ze vstupních Bytových polí souborů popisující síť. Také je zde zabudováno použití algoritmu NMS [32] a filtrace detekovaných průstřelů na základě minimální možné jistoty každého průstřelu. Výstup této komponenty je list ohraničujících obdélníků nalezených průstřelů.

- **Hodnotitel průstřelů** - Poslední komponenta má na starost hodnocení průstřelů. Přijímá list ohraničujících obdélníků nalezených průstřelů. Informace o těchto obdélnících pak spolu s definovanou šablonou pro daný terč komponenta využívá k automatickému obodování střelce. Výstup výpočtu této komponenty pak tedy je celočíselná hodnota určující kolik bodů střelec nastřílel a terč s nalezenými označenými průstřely.




Obrázek 28: Základní C++ rozhraní aplikace

.

## 6.3 Popis uživatelského rozhraní aplikace

Uživatelské rozhraní v aplikaci je velice jednoduché a prozatím slouží spíše pro testování automatického hodnocení průstřelů.

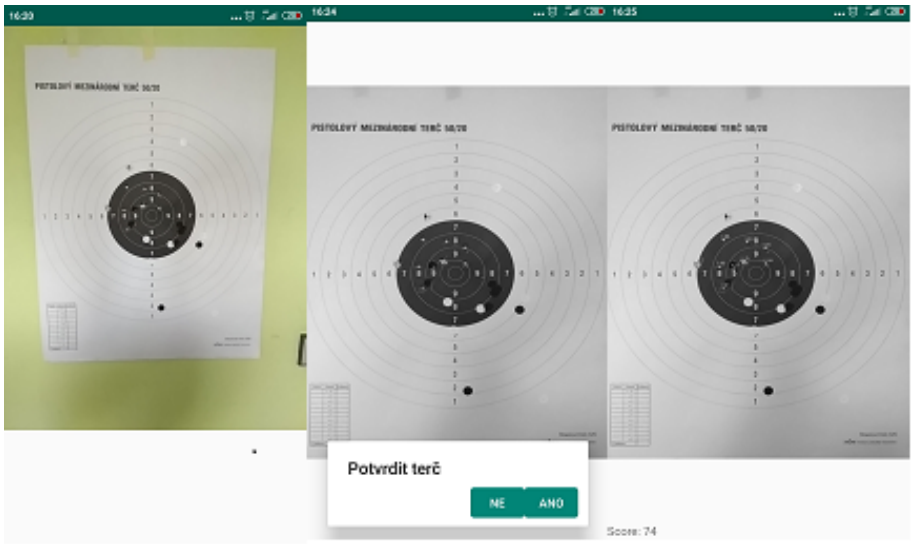
Po spuštění aplikace se uživateli objeví na obrazovce tlačítko pro spuštění celé funkcionality automatického bodování. Po stisknutí tlačítka se spustí kamera na telefonu spolu se světelným zdrojem umístěným vedle kamery (pokud jím telefon disponuje). Osvětlení pomáhá eliminovat možný částečně zastíněný terč. Toto zastínění by mohlo mít vliv na nalezení rohů terče, potřebných pro perspektivní transformaci.

Po spuštění Android aktivity obsluhující kameru se automaticky začnou posílat jednotlivé snímky z kamery do C++ části aplikace pro zpracování. Jelikož vyhledání terče a jeho transformace nějakou chvíli trvá, neposílám na zpracování každý snímek z kamery, ale pouze každý desátý. Toto nijak neovlivní pohled uživatele na náhled kamery na obrazovce, jelikož zpracování snímku probíhá asynchronně na jiném vláknu aplikace.

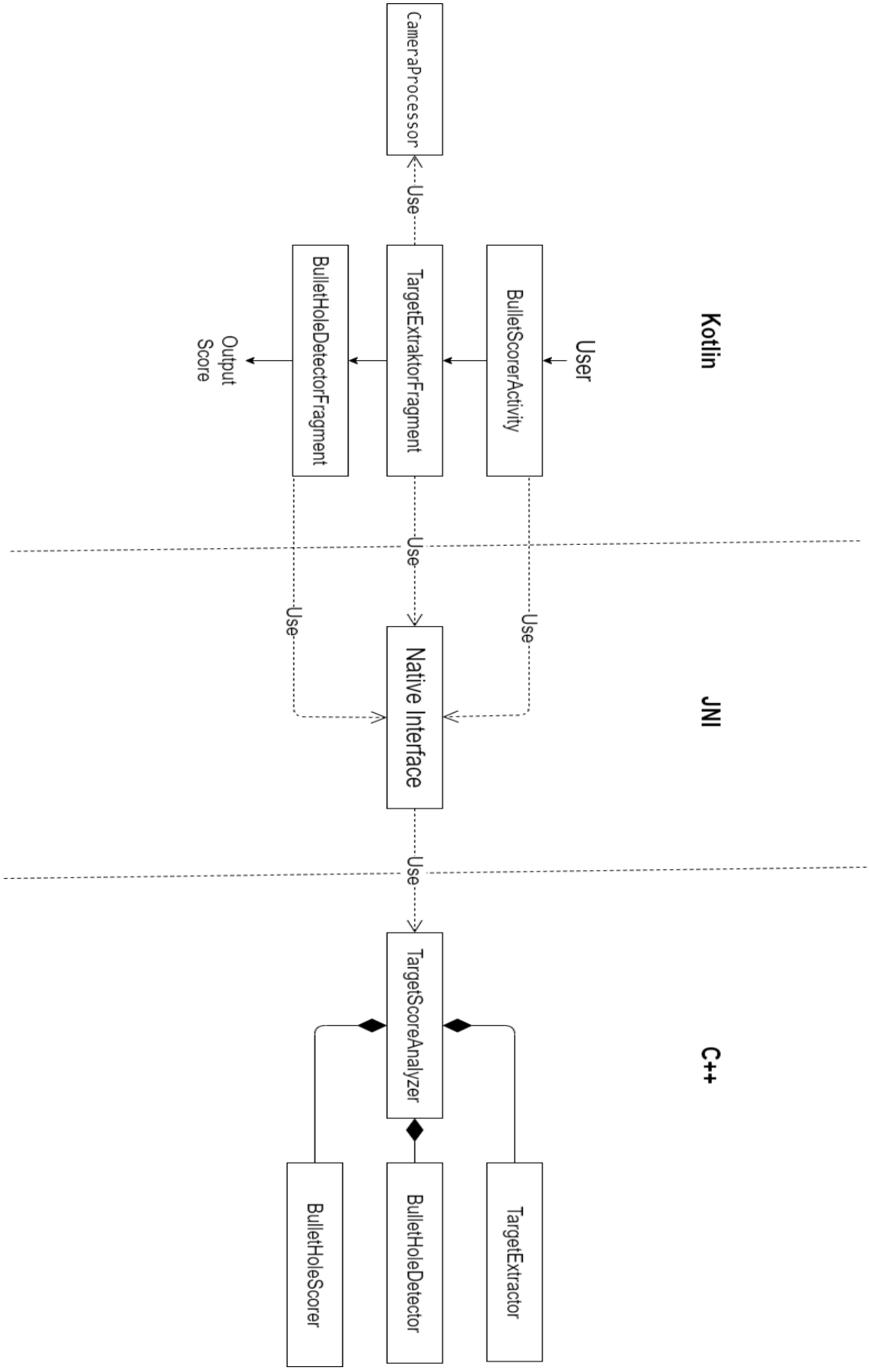
Pokud Extraktor terče nedokáže terč ve snímku najít, uživatel se o tom nijak nedozví a kamera jednoduše zašle další snímek. V opačném případě se zastaví kamera, vypne se světelný zdroj a spustí se jiný pohled aplikace s výsledným obrázkem transformovaného terče. Aplikace se uživatele zeptá, jestli obrázek, který vidí, chce dále zpracovávat (obrázek může být rozmazaný nebo zde teoreticky může být zachycen jiný objekt než terč).

Pokud se uživatel rozhodne, že obrázek není dost dobrý na další zpracování, aplikace přejde zpět do fáze extrakce terče pomocí kamery a celý postup se opakuje. V opačném případě se extrahovaný obrázek pošle na další zpracování (detektor průstřelů a hodnotitel). Po zpracování se otevře nový náhled aplikace s obrázkem výsledného terče a jednoduchý text s počtem bodů za detekované průstřely.

Na obrázku 29 můžeme vidět všechny pohledy jednoduchého uživatelského rozhraní.




Obrázek 29: Vizualizace jednoduchého uživatelského rozhraní




Obrázek 30: Vizualizace základní architektury aplikace

# 7 Závěr

V diplomové práci byl položen základ pro střeleckou aplikaci, která by měla být v budoucnu jednoduše rozšiřitelná na více mobilních operačních systémů a práci s různými typy terčů. Část aplikace starající se o extrakci terče z obrazu by navíc mohla být s mírnými úpravami znovupoužitelná v aplikaci jiného typu, kde je potřeba vysegmentovat nějaké obdélníkové objekty z obrazu (například scanner dokumentů). Z výsledků YOLOv3 tiny neuronové sítě [22] na testovací datové sadě, vyplývá, že tento způsob detekce průstřelů se nezdá být slepá cesta a po dostatečném zvětšení trénovací datové sady by mohla fungovat v reálném prostředí s přijatelnou úspěšností. Aplikace zatím není připravena na masové použití v reálném prostředí, hlavně kvůli následujícím problémům.

- Hlavním nedostatkem je malá trénovací datová sada, použita pro natrénování konvoluční neuronové sítě. Síť má docela dobrou úspěšnost na trénovacích a testovacích snímcích (trénovací i testovací datavé sady byly pořízeny v podobných světelných podmínkách), nicméně trénovací ani testovací sada zdaleka nepokrývá všechny možné variace vzhledu průstřelů při různém osvětlení a tím pádem může jednoduše selhat na nenatrénovaných snímcích s naprosto odlišným osvětlením. Toto je asi největší problém, jelikož aby chtěl uživatel vůbec aplikaci používat musel by být detektor (podle mého názoru) alespoň z 95% bezchybný na naprosto náhodně vybraných datech z reálného prostředí, jinak by takové aplikaci nedůvěřoval a raději si body počítal ručně.

  Řešením tohoto problému je docela jasné, a to rapidně zvětšit trénovací datovou sadu tak, aby reprezentovala většinu možných tvarů a osvětlení průstřelů. Nabízí se zde také uměle navýšit sadu tím, že bych ze základní sady vygeneroval mnohem větší sadu, která by obsahovala původní snímky spolu s nějak náhodně upravenými snímky (například rotace průstřelu nebo rozmazání).

- Dalším problémem může být pro uživatele nepříliš rychlý detektor průstřelů, který jeden snímek dokáže zpracovat průměrně za 6 vteřin. Toto by mohlo některé uživatele odrazovat od použití.

  Pro zrychlení detektoru bych mohl vyzkoušet jiné knihovny s implementací Konvolučních neuronových sítí, které jsou přímo určeny a optimalizovány na mobilní zařízení, což OpenCV [29] bohužel není. Toto by mohlo ale také nemuselo fungovat a bylo by to třeba otestovat. Také se nabízí možnost celý výpočet přemístit na nějaký server s mnohem větším výpočetním výkonem a komunikovat s ním přes internet. Toto řešení má bohužel jeden velký nedostatek, kterým je absence signálu v mobilním telefonu na střelnici, některé střelnice jsou obehnány hrubými zdmi a stropem, blokující signál.

- Nevýhoda detektoru také je, že kraje terče musí být zřetelně rozeznatelné od pozadí, a tím pádem terč s pozadím bílé stěny vedle něho nebo za ním může být problém. Tento případ

se naštěstí neděje moc často, protože za terčem je většinou mnohem tmavší okolí, jelikož na střelnicích je pozadí většinou dost vzdáleno od terče (K tomu nám pomáhá i osvětlení terče pomocí světelného zdroje na telefonu).

Pro tento problém mě nenapadá žádné lepší řešení, jelikož většina segmentačních algoritmů je založena na informaci o hranici objektu. Pokud se tato hranice ztratí, algoritmus nemůže fungovat dobře. Naštěstí tento problém není zase tak podstatný, z důvodů, které jsem popisoval výše.

Na základě dat z aplikace (body za průstřely a jejich pozice v terči) by se dal vybudovat celý informační systém, který by schraňoval veškerá historická data o střelci a jeho průstřelech. Na základě těchto dat by mu případně mohla aplikace i radit v tom, co nejspíš dělá při střelbě špatně (informace o průměrném rozptylu od středu terče). Také by mohli vzniknout online žebříčky, kde by se střelci mohli navzájem porovnávat, což by mohlo vést ke větší motivaci se ve střelbě zlepšit.

# Literatura


[1] Box blur filter [online]. [cit. 2020-03-11].
Dostupné z: https://en.wikipedia.org/wiki/Box_blur

[2] HSV image representation [online]. [cit. 2020-03-11].
Dostupné z: https://en.wikipedia.org/wiki/HSL_and_HSV

[3] J. Canny, A Computational Approach to Edge Detection. in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, Nov. 1986, pp. 679-698, doi: 10.1109/TPAMI.1986.4767851.

[4] Srisha, Ravi & Khan, Am. Morphological Operations for Image Processing : Understanding and its Applications, 2013.

[5] Flood fill algorithm [online]. [cit. 2020-03-11].
Dostupné z: https://en.wikipedia.org/wiki/Flood_fill

[6] SOJKA, Eduard. Digitální zpracování a analýza obrazů. Ostrava: VŠB-Technická univerzita, chapter 9.11, 2000. ISBN isbn80-7078-746-5

[7] Perspective transform [online]. [cit. 2020-03-15].
Dostupné z: https://www.sciencedirect.com/topics/computer-science/perspective-transform

[8] Tested mobile phone [online]. [cit. 2020-03-17].
Dostupné z: https://www.gsmarena.com/xiaomi_redmi_4_(4x)-8608.php

[9] Joseph Redmon. Darknet: Open Source Neural Networks in C.
http://pjreddie.com/darknet/, 2013–2016.

[10] Bounding box annotation tool [online]. [cit. 2020-03-18].
Dostupné z: https://mc.ai/boobs%E2%80%8A-%E2%80%8Ayolo-bbox-annotation-tool

[11] J. Redmon, S. Divvala, R. Girshick and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 779-788

[12] O'Shea, Keiron & Nash, Ryan. An Introduction to Convolutional Neural Networks. ArXiv e-prints, 2015

[13] Darknet 53 model [online]. [cit. 2020-03-19].
Dostupné z: https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193

[14] Everingham, M. and Eslami, S. M. A. and Van Gool, L. and Williams, C. K. I. and Winn, J. and Zisserman, A. The Pascal Visual Object Classes Challenge: A Retrospective. International Journal of Computer Vision 111, 2015, pp. 98–136.

[15] R. Girshick, et al. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014.

[16] R. Girshick. Fast r-cnn. In ICCV, 2015, pp. 1440-1448.

[17] Evgeniou T., Pontil M. Support Vector Machines: Theory and Applications. In: Paliouras G., Karkaletsis V., Spyropoulos C.D. (eds) Machine Learning and Its Applications. ACAI 1999. Lecture Notes in Computer Science, vol 2049. Springer, Berlin, Heidelberg, 2001.

[18] Ren, Shaoqing et al. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. IEEE Transactions on Pattern Analysis and Machine Intelligence 39, 2015.

[19] Coco dataset[online]. [cit. 2020-03-21].
Dostupné z: http://cocodataset.org

[20] Beitzel S.M., Jensen E.C., Frieder O. Mean Average Precision. In: LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA, 2009.

[21] Redmon, Joseph, and Ali, Farhadi. YOLOv3: An Incremental Improvement, 2018.

[22] Wangpeng He *, Zhe Huang, Zhifei Wei, Cheng Li and Baolong Guo. TF-YOLO: An Improved Incremental Network for Real-Time Object Detection. Applied Sciences 9, 2019.

[23] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), San Diego, CA, USA, 2005, pp. 886-893 vol. 1, doi: 10.1109/CVPR.2005.177.

[24] King Davis. Max-Margin Object Detection, 2015.

[25] Davis E. King. Dlib-ml: A Machine Learning Toolkit. Journal of Machine Learning Research 10, 2009, pp. 1755-1758.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[27] Vanishing gradients problem [online]. [cit. 2020-03-24].
Dostupné z: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

[28] Pooling layers [online]. [cit. 2020-03-25].
Dostupné z: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks

[29] Bradski G., The OpenCV Library, Dr. Dobb's Journal of Software Tools, 2000.

[30] Mean average precision [online]. [cit. 2020-04-06].
Dostupné z: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

[31] Intersection over union [online]. [cit. 2020-04-06].
Dostupné z: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection

[32] Navaneeth Bodla, Bharat Singh, Rama Chellappa, Larry S. Davis. Improving Object Detection With One Line of Code. Center For Automation Research, University of Maryland, College Park, 2017.

[33] Android OS [online]. [cit. 2020-04-20].
Dostupné z: https://en.wikipedia.org/wiki/Android_(operating_system)

[34] Kotlin programming language [online]. [cit. 2020-04-20].
https://en.wikipedia.org/wiki/Kotlin_(programming_language)

[35] Java programming language [online]. [cit. 2020-04-20].
Dostupné z: https://en.wikipedia.org/wiki/Java_(programming_language)

[36] Java Virtaul Machine [online]. [cit. 2020-04-20].
Dostupné z: https://cs.wikipedia.org/wiki/Java_Virtual_Machine

[37] Android native development kit [online]. [cit. 2020-04-20].
Dostupné z: https://developer.android.com/ndk/guides

[38] Java native interface [online]. [cit. 2020-04-21].
Dostupné z: https://en.wikipedia.org/wiki/Java_Native_Interface

[39] Parkour police target[online]. [cit. 2020-05-13].
Dostupné z: https://www.armyeshop.cz/1353-detail-strelecke-terce-zalepky-terc-policejni-parkur-660x500mm-10ks

# Přílohy

Příloha práce obsahuje .zip soubor, který se skládá ze dvou adresářů:

(a) adresář s testovací konsolovou aplikací pro jednoduché testování (název: ShootingRange-AppCore) - Obsahuje trénovací i testovací datovou sadu (adresáře: target_test_dataset a target_train_dataset), dále je zde několik neextrahovaných terčů ve složce target_samples. Adresář obsahuje dva soubory popisující YOLO neuronovou síť, kterými jsou yolov3-tiny-obj.cfg a yolov3-tiny-obj_best.weights.

(b) adresář se samotnou mobilní aplikací (název: shoot) - Zdrojové soubory jsou pak obsaženy v shoot/app/src/main složce.

Soubor .zip dále obsahuje soubor readme.txt, který popisuje, jak aplikaci nainstalovat a spustit na mobilním telefonu nebo emulátoru. Také je zde popis jak používat testovací aplikaci.